Universitat de Lleida
Escola Politècnica Superior
Ingeniería Técnica en Informática de Sistemas

Trabajo de final de carrera

Genereitor 2

Generador de código J2EE

Autor: Hugo Alonso Capuz
Director: Ramón Béjar Torres
Septiembre de 2008

Agradecimientos

Durante la realización de este proyecto, muchas han sido las personas que, de una forma u otra, me han aportado su ayuda, su apoyo y sus ánimos. En especial, he de dar las gracias...

...a mis padres, por brindarme la oportunidad de estudiar una carrera, por celebrar los aprobados y por comprender los suspensos, siempre con una sonrisa en los labios.

...a Andrea, por apoyarme, por animarme y por estar ahí siempre que la necesitaba.

...a Raúl, porque sin su ayuda hubiera sido mucho más duro.

...a Juan y Nacho, por confiarme la realización de este proyecto y abrirme las puertas del mundo laboral.

...a Ramón, por ofrecerse a ser mi tutor y darme consejo y opinión.

...a mis compañeros de la oficina, por alegrar las mañanas de lunes y compartir caras de sueño.

A todos vosotros... ***¡Gracias!***

Índice general

Índice de figuras

Parte I

Introducción

Capítulo 1

Situación inicial

1.1. Aplicaciones empresariales

Hoy en día cualquier empresa de cierta envergadura necesita manejar sus datos de forma sencilla, rápida y eficaz. Hace años, toda la información se almacenaba en archivos de papel, lo que ocasionaba infinidad de problemas a la hora de hacer cualquier consulta o tratamiento de datos: los archivadores de papel ocupan mucho tamaño, la consulta, modificación o inserción de datos cuestan mucho tiempo y el mantenimiento es complicado, al ser un soporte físico susceptible al desgaste o a la destrucción por accidentes (fuego, inundaciones, etc). Asímismo, las copias de seguridad son complicadas de hacer, ya que duplicar todos los registros de un archivo de cierta magnitud conlleva un gran coste tanto económico como de tiempo, al márgen de la necesidad de disponer de espacio en otro lugar para albergar la copia. También mantener y actualizar un archivo de seguridad implica más trabajo y más tiempo.

Con la aparición de la informática estos problemas comenzaron a desaparecer. Cuando los sistemas de bases de datos se desarrollaron, se convirtieron en una ventajosa alternativa frente al tradicional soporte de papel, ofreciendo rapidez y comodidad, al tiempo que desaparecían los problemas de tamaño. La cinta magnética se convirtió en el soporte ideal para albergar copias de seguridad de bancos de datos de gran magnitud, ya que pese a su lentitud en el acceso a los datos (aún así era infinitamente más rápido que las copias en papel), eran un soporte económico y eficaz.

Conforme la tecnología avanzaba, la demanda de sistemas de almacenamiento de mayor calidad y facilidad de uso incrementó enormemente, y en este marco es donde surge el concepto de *aplicación empresarial*. Una aplicación empresarial es, a grandes rasgos, un sistema informático que ayuda a incrementar o controlar la productividad de una empresa.

El principal objetivo de una aplicación empresarial es incrementar los beneficios de una empresa mediante la reducción de costes o la aceleración del ciclo productivo, delegando a una máquina la realización de tareas que antes requerían del trabajo de varios empleados.

1.2. Aplicaciones web

La implementación de una aplicación empresarial se puede enfocar de diversas maneras, pero actualmente las aplicaciones web son la forma más eficiente de llevar a cabo un proyecto de este tipo.

Una definición de *aplicación web* podría ser un sistema a los que los usuarios acceden a través de un servidor web mediante Internet o una intranet[1], utilizando para ello un navegador. La aplicación está codificada en lenguajes soportados por los navegadores web (HTML, JavaScript, etc).

Este tipo de aplicaciones son populares debido a lo práctico del uso del navegador web como cliente ligero, puesto que prácticamente todos los sistemas informáticos cuentan en su configuración por defecto con uno, lo que supone una gran ventaja a la hora de llevar a cabo actualizaciones o mantenimiento de la aplicación, puesto que que estas tareas se llevan a cabo en el servidor. Esto presenta grandes ventajas:

- Las modificaciones no tienen que ser instaladas y ejecutadas en cada uno de los clientes o usuarios, lo que simplifica enormemente las tareas de actualización.
- Es sencillo implementar nuevas funcionalidades (tales como procesadores de texto, clientes de correo o motores de búsqueda) e integrarlas en la aplicación.
- No se necesita espacio en disco exclusivo para la aplicación en la máquina del cliente, dado que no existe ningún proceso de instalación.
- Proporciona una independencia de plataforma respecto del cliente, no importando el sistema operativo o el navegador que éste utilice[2].

Por contra, también adolecen de una serie de inconvenientes respecto de las aplicaciones *de escritorio*:

- Dependen de una conexión con Internet o la intranet donde se aloja la aplicación, por lo tanto si esta conexión se ve interrumpida, la aplicación no es accesible.

[1] Una **intranet** es un conjunto de contenidos compartidos por una organización, o un grupo determinado de componentes de ésta. Es una potente herramienta que permite divulgar información de la compañía a sus empleados con efectividad, estando éstos informados de las últimas novedades y datos de la organización.
Su función principal es proveer lógica de negocios para aplicaciones de captura, informes y consultas con el fin de facilitar la producción de los grupos de trabajo. También es un importante medio de difusión interna de la información propia de la compañía.

[2] Aunque una aplicación web proporciona independencia de plataforma, una implementación inconsistente de HTML, CSS, DOM u otras especificaciones del navegador puede interferir en el funcionamiento de la aplicación. Además, las características propias de los navegadores, tales como la posibilidad de modificar el tamaño de fuente o deshabilitar la ejecución de scripts, pueden afectar a su funcionamiento.

- Las funciones de interfaz que puede ofrecer un navegador web son mucho más limitadas que aquellas de las que se disponen en una aplicación de escritorio. Por ejemplo, la característica *arrastrar y soltar*, aunque es posible disponer de ella en una aplicación web, es mucho más difícil de implementar que en una aplicación de escritorio.
- La amplia difusión de la que gozan las aplicaciones de escritorio en ciertos campos (e.g. tareas de ofimática) supone una traba para las aplicaciones web de este tipo, ya que es complicado ofrecer compatibilidad con los contenidos creados inicialmente con aquellas.

Una aplicación web generalmente contiene elementos que permiten interactividad entre el usuario y la información, tales como el acceso a bases de datos o la cumplimentación de formularios.

1.3. Ventajas perseguidas con el desarrollo de Genereitor

Las soluciones de software empresarial ofrecidas por los fabricantes de forma «genérica» no siempre satisfacen las necesidades de las empresas, hecho que se acrecenta en corporaciones de cierta magnitud. Es por ésto que muchas veces se opta por prescindir de estas soluciones e implementar las propias, de forma personalizada para que cubran exactamente las necesidades de cada caso.

Así, aparecen empresas dedicadas a desarrollar soluciones informáticas personalizadas para cada cliente, de forma que tras un estudio de las necesidades y los requisitos del cliente, se le ofrezca un producto totalmente personalizado y adecuado a su negocio.

Las ventajas de una aplicación personalizada frente a un paquete *de fábrica* son innumerables, dado que desde el principio del desarrollo del producto se tiene en cuenta la situación específica del cliente a quien va dirigida, en lugar de tener que usar un paquete prefabricado y aprovechar las características que le sean útiles a la empresa. En muchas ocasiones el uso de estos paquetes provoca, aparte de que muchas de las funciones queden desaprovechadas, que la propia empresa tenga que modificar su metodología de trabajo para adaptarse al software de gestión.

Comex Grupo Ibérica es una empresa del sector de las TIC, y uno de sus cometidos es el desarrollo de aplicaciones empresariales a medida. Desarrollar una aplicación de este tipo requiere invertir mucho tiempo y esfuerzo, lo que en consecuencia deriva en dos de los grandes inconvenientes de las soluciones de software a medida: el **tiempo** de desarrollo y derivado de esto, el **precio** final del producto. Con Genereitor se persiguen las siguientes metas:

Ahorro de tiempo. Si se dispone de una aplicación *a medio hacer* y acorde con las necesidades y características del proyecto al que se enfrenta el equipo de desarrollo, la fase inicial de la implementación es mucho más directa.

Al disponer ya de versiones «primitivas» de todos los niveles de la aplicación, desde interfaz web hasta métodos de acceso a datos, el desarrollo comienza con la implementación de las partes específicas de la aplicación, haciendo uso de la pericia y creatividad de los programadores para tareas específicas de cada proyecto que dificilmente podrían ser automatizables.

Optimización del código. Delegando en Genereitor la implementación de las bases de un proyecto, se garantizan unos cimientos sólidos, ya que el código generado se basará en unas plantillas que han sido revisadas minuciosamente, y que conforme se vayan descubriendo fallos en las mismas, se implementen nuevas mejoras o se mejore su eficiencia, éstos se incorporarán a todos los proyectos. Al trabajar siempre sobre bases similares, se garantiza que el código generado sea revisado indirectamente muchas más veces que el que pueda implementar una persona, por lo que los fallos se detectarán mucho antes.

Homogeneidad dentro del proyecto. En la implementación de proyectos de cierta envergadura es normal que trabajen varias personas, que pueden tener diferentes costumbres a la hora de programar. Mientras uno describa los métodos con nombres más descriptivos, otro lo hará con literales más concisos y abreviados. Esta situación se convierte en un problema a la hora de realizar el mantenimiento o la resolución de incidencias de un proyecto, puesto que se entremezclan los diferentes criterios de los distintos programadores que han implementado las diferentes partes de la aplicación, teniendo que consultar cómo se ha llamado a cada método o, en el peor de los casos, modificar los nombres de estos métodos –con las consecuencias que ésto puede acarrear, como que otras partes del proyecto dejen de funcionar correctamente–.

Genereitor proporciona un único criterio para el nombrado de variables, constantes, métodos, identificadores de campos en los formularios web, etc., de forma que el mantenimiento o la implementación de funciones nuevas es más cómodo, puesto que el programador siempre va a conocer cómo se han nombrado los métodos o los campos del formulario que tiene que utilizar, sin necesidad de consultarlo en la documentación o en la especificación de la clase.

Homogeneidad entre proyectos. El uso de Genereitor implica que todos los proyectos que se desarrollen con su asistencia van a tener la misma estructura, tanto a nivel lógico como de directorios, lo que permite agilizar el trabajo al programador que haya adquirido un mínimo de costumbre, puesto que no tendrá que buscar por el árbol de directorios dónde está cierto componente o parámetro de configuración: siempre estarán en el mismo lugar. Más adelante se detallará esta estructura de directorios.

Documentación. En el ámbito empresarial, el desarrollo de las aplicaciones siempre se ve obstaculizado por las prisas. Perder más tiempo de la cuenta en el desarrollo de un proyecto significa disminuir el márgen de beneficios, y a consecuencia de ésto los plazos suelen ser muy ajustados.

Por esta razón, nunca suele haber una buena documentación para cada aplicación que se implementa, puesto que siempre hay cosas más urgentes en las que emplear el tiempo.

Así, Genereitor ofrece en el código generado comentarios explicativos de las partes menos claras, así como notas en formato Javadoc para la generación automática de documentación, que podrá ser consultada tanto vía web (de forma similar a la consulta de las APIs de J2EE en la página de Sun) como en las notas emergentes de los diferentes entornos integrados de desarrollo que se utilizan (NetBeans, Oracle Jdeveloper, etc).

Evidentemente no es posible ofrecer de forma automática una explicación aclarativa del funcionamiento y la finalidad de cada método implementado, pero sí se ofrecen las bases de esta documentación, de manera que el programador pueda rellenar en el código de forma breve las funciones que crea convenientes, obteniendo en poco tiempo y *sobre la marcha* una documentación completa que ayude a quien en un futuro haya de realizar un mantenimiento de la aplicación.

Un ejemplo de la documentación ofrecida podría ser el siguiente:

```
1       /**
2        *
3        * @param conn Connection
4        * @param filtro Vehiculo
5        * @param orderBy String
6        * @return com.comex.common.bean.BaseBeanSet
7        * @throws java.sql.SQLException
8        */
9        public static BaseBeanSet getListaVehiculos(Connec...
```

A partir de estos comentarios, Javadoc es capaz de generar la documentación necesaria.

La finalidad de Genereitor consiste, a grandes rasgos, en proporcionar al desarrollador la estructura de la aplicación totalmente personalizada conforme a los requisitos del proyecto (nombre de la aplicación y los paquetes y clases que la componen, adaptación conforme al servidor de aplicaciones utilizado y sistema de bases de datos sobre el que operará), ya preparada para comenzar el desarrollo, evitando así la tarea de crear dicha estructura a mano.

Asímismo, se proveerán scripts de compilación para la herramienta ant conforme a los diferentes componentes, dependencias y librerías del proyecto. Se proporcionarán dos tipos de scripts:

- Scripts para desarrollo.
- Scripts para preproducción y producción, orientados a la compilación y despliegue en los servidores del cliente.

También se proporcionan, para cada entidad de la base de datos, paquetes que contendrán el código fuente correspondiente a todas las capas de la aplicación (tanto capa de presentación como lógica de negocio), que podrán ser *insertadas* en la aplicación como si de módulos se tratase, de manera que se puedan añadir a la aplicación las funciones correspondientes a cada entidad de forma progresiva conforme avance la implementación.

La conjunción de la estructura básica generada al principio con los módulos creados a posteriori darán como resultado una aplicación web completa que,

aunque de forma básica y primitiva, será totalmente funcional, incluyendo una rudimentaria interfaz web y funciones de lectura, inserción, modificación y borrado de todo tipo de entidades existentes en la base de datos.

Así, el programador obtiene una aplicación que ya funciona sin tocar una sola línea de código, sobre la que comenzar a trabajar. Evidentemente esto supone un ahorro de tiempo enorme comparado con lo que costaría empezar el proyecto desde cero, teniendo que implementar todo *a mano*.

1.4. Requisitos de Genereitor

Genereitor es una herramienta de generación de aplicaciones web automática, que mediante la conexión a una base de datos previamente implementada y los requisitos de la aplicación a generar es capaz de devolver al programador dicha aplicación en una fase muy temprana del desarrollo, aunque completamente funcional. Así, los requisitos básicos son los que se listan a continuación:

1.4.1. Aplicación web

Genereitor se implementará como una aplicación web, cuyo acceso se efectúe mediante un navegador web estándar. Aunque no es indispensable, se ha decidido implementar mediante la tecnología J2EE por diversos motivos:

- Es la tecnología que se utilizará obligatoriamente para las aplicaciones generadas con Genereitor, por lo que servirá de toma de contacto, aprendizaje y experiencia para el posterior desarrollo de las plantillas que generarán dichas aplicaciones.
- Es una tecnología madura y ampliamente extendida en el campo de las aplicaciones web, ofrece una estructura sólida y modular y dispone de multitud de APIs y componentes a disposición del programador, de forma que ahorra mucho trabajo al no tener que implementar dichos componentes, puesto que están integrados en el servidor de aplicaciones.
- Su aprendizaje no es excesivamente difícil, por lo que se prefiere a otras tecnologías más complicadas.
- Es una tecnología libre y puede ser desarrollada con herramientas completamente gratuítas, lo que supone una ventaja a la hora de incorporar Genereitor a producción, ya que se evitan gastos de licencias de uso que se producen al utilizar otras tecnologías tales como .Net.

1.4.2. Conexión a bases de datos

La herramienta ha de ser capaz de generar aplicaciones adaptadas a diversos sistemas de bases de datos, que variarán de un proyecto a otro. Es por esto que debe ser capaz de manejar eficientemente y generar las capas de acceso a datos para la base de datos contra la que se apoyará la aplicación a generar.

Inicialmente, los sistemas de bases de datos soportados son:

- Oracle 9
- PostgreSQL
- MySQL
- Microsoft SQLServer
- HSQLDB

1.4.3. Integración de los módulos de la empresa

En la empresa se utilizan diversos módulos que han sido desarrollados a lo largo del tiempo y que ofrecen funcionalidades específicas a los proyectos, tales como com.comex.cms, que proporciona un sistema de gestión de contenidos, com.comex.xml que maneja el envío y recepción de mensajes mediante XML o com.comex.usuarios, que da soporte para la gestión de usuarios, perfiles y grupos a una aplicación web.

Genereitor será capaz de integrar, bajo demanda del programador, estos módulos a la aplicación a generar, personalizando su comportamiento y modificando los ficheros de configuración para compilar automáticamente los ficheros fuente de dichos módulos.

1.4.4. Tratamiento de relaciones entre entidades

Genereitor ha de ser capaz de averiguar las relaciones que tienen las diferentes entidades de una base de datos, informando de ello al programador y ofreciendo la posibilidad de ser tratadas en la generación de aplicaciones, de forma que dichas aplicaciones ya incorporen en su lógica el tratamiento de las claves ajenas.

1.4.5. Interfaz intuitiva

Genereitor contará con una interfaz clara e intuitiva, con literales que identifiquen claramente la función de cada uno de los componentes que forman parte de ésta. Cuando, por cuestiones de espacio, sea difícil ofrecer literales demasiado específicos, se proveerán de explicaciones mediante etiquetas flotantes al pasar el puntero del ratón sobre dicho literal.

Asimismo, el manejo de errores será capaz de informar al usuario de cuándo ha ocurrido algún fallo en la ejecución del programa o la introducción de datos, indicando la causa del mismo.

1.5. Requisitos de las aplicaciones desarrolladas con Genereitor

Las aplicaciones desarrolladas con Genereitor serán aplicaciones web implementadas en J2EE[1], cuya estructura seguirá las recomendaciones de las Java BluePrints[2].

Permitirán tanto la consulta como la modificación de los datos almacenados en una base de datos, que podrá estar soportada por diferentes servidores de bases de datos.

Según las necesidades del proyecto, la capa de aplicación podrá estar implementada mediante EJBs o POJOs, y en consecuencia serán desplegables en diversos servidores de aplicaciones, con o sin contenedor de EJB.

Las interfaces generadas serán accesibles a nivel AA[3], aunque se ofrecerán soluciones más eficientes cuando este nivel de accesibilidad no sea necesario.

Serán compatibles con los módulos desarrollados en la empresa (com.comex.common, com.comex.tablasmaestras, com.comex.cms, com.comex.usuarios, com.comex.xml, etc).

Han de ser parametrizables en cuanto al despliegue en diferentes entornos, que corresponderán a las distintas etapas de desarrollo de la aplicación. Los scripts de despliegue proporcionados aceptarán parámetros que modificarán el despliegue según se encuentre el proyecto en *desarrollo*, *preproducción* o *producción*.

El tratamiento de archivos binarios se llevará a cabo mediante *streaming*, tanto para la carga como la descarga de grandes binarios al servidor de bases de datos, lo que evitará la saturación del servidor de aplicaciones por falta de memoria, a la vez que facilita el manejo de archivos multimedia en caso de que se quiera añadir un reproductor de medios.

Seguirán los siguientes patrones de desarrollo:

Modelo-Vista-Controlador. Modelo-Vista-Controlador (Model-View-Controller) es un patrón de desarrollo que separa la parte lógica de una aplicación de su presentación. Básicamente sirve para separar el lenguaje de programación del HTML lo máximo posible y para poder reutilizar componentes fácilmente.

El Modelo representa las estructuras de datos. Típicamente el modelo de clases contendrá funciones para consultar, insertar y actualizar información de la base de datos.

La Vista es la información presentada al usuario. Una vista puede ser una página web o una parte de una página.

El Controlador actúa como intermediario entre el Modelo, la Vista y cualquier otro recurso necesario para generar una página.

Filtro. El patrón filtro consiste en el uso de objetos con atributos incompletos –que no serán válidos para el tratamiento de datos–, pero que se utilizan para la recogida de datos en formularios y en las operaciones de lógica de

la interfaz, que mediante la consulta al modelo de datos completarán los atributos del filtro para crear un objeto válido.

Facade. La fachada se usa para proporcionar una interfaz para la capa de lógica de negocio, de forma que los demás componentes de la aplicación accedan a dicha capa a través de la fachada, en lugar de hacerlo directamente. Para ello, se implementa un bean de fachada, que representa los componentes de la lógica de alto nivel, que a su vez interactuarán con los componentes de bajo nivel.

El uso de fachada repercute en una mayor facilidad de uso y legibilidad de la capa de lógica de la aplicación, al tiempo que flexibiliza el desarrollo.

Singleton. Este patrón restringe la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto.

Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

El patrón singleton se implementa creando en nuestra clase un método que crea una instancia del objeto sólo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor.

ServiceLocator. Los componentes J2EE se vale de APIs que proporcionan diferentes servicios, tales como interfaces EJB, DataSources, conexiones, etc. El patrón ServiceLocator permite proveer a la aplicación de un controlador centralizado que permite manejar todos estos servicios desde un mismo componente, evitando el tener que buscar a lo largo de toda la aplicación los servicios necesarios en cada momento.

Capítulo 2

Antiguo generador

Antes del desarrollo de Genereitor ya existía una herramienta de características similares, pero sus funciones era muy limitadas. En lugar de generarse un proyecto completo y funcional, se creaban únicamente los cimientos de varios ficheros de código, que luego habrían de ser completados y adaptados por el programador.

La antigüedad de esta herramienta y la dificultad de su mantenimiento propició que cayera progresivamente en desuso, puesto que el ahorro de tiempo que suponía al principio fue mermando con el paso del tiempo, al evolucionar las características y los requisitos de los proyectos demandados por los clientes. La falta de mantenimiento regular se tradujo en que, actualmente, cuesta más arreglar y adaptar el código generado que implementarlo desde cero o reciclar un proyecto existente y adaptarlo a mano. Además, las escasas funciones que aún son aprovechables no compensan con el esfuerzo invertido en la adaptación a los nuevos métodos de trabajo.

A continuación se detallan los problemas más importantes de los que adolece el generador actual.

2.1. Carencias

2.1.1. Relaciones entre entidades

La característica principal de las bases de datos relacionales es, precisamente, las relaciones entre las diferentes entidades de la misma. Estos lazos entre las entidades se representan mediante claves ajenas, que son características comunes y compartidas por dos entidades.

Así, es ineludible el tratamiento de estas relaciones de manera eficiente por cualquier aplicación que explote una base de datos, siempre teniendo en cuenta las restricciones de ésta, de forma que se mantenga la consistencia de la información almacenada.

El antiguo generador sólo tenía en cuenta las características propias de cada entidad, es decir, sólo proporcionaba la lógica para el tratamiento de los campos propios de cada tabla, de manera que las relaciones con otras tablas había que hacerlas a mano, tarea que conlleva tiempo y esfuerzo. Otro inconveniente es que, al tenerlas que hacer una por una, el código implementado por el programador es susceptible de fallos que hagan que la aplicación no funcione correctamente en determinadas ocasiones que, por extrañas o inusuales, no se hayan tenido en cuenta inicialmente por el programador, o que en el peor de los casos, pongan en peligro la integridad de la base de datos si ésta no está perfectamente diseñada.

Por esto, Genereitor tiene en cuenta automáticamente las relaciones entre tablas, ofreciendo la lógica para el tratamiento de las mismas, de la forma que indica la figura 2.1.

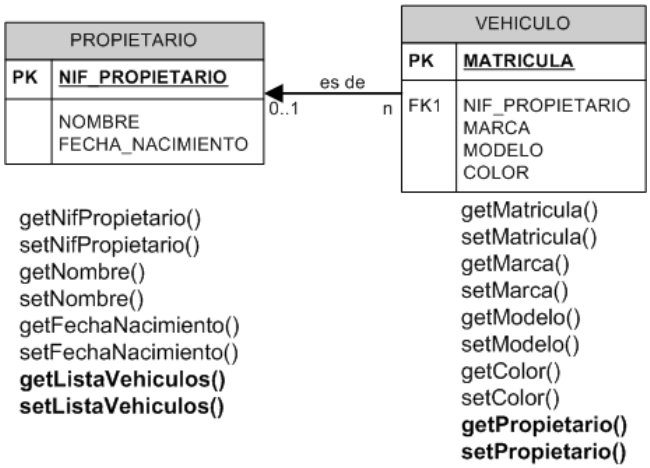

Figura 2.1: Métodos que implementa Genereitor para dos entidades relacionadas

2.1.2. Interfaz

En el desarrollo del antiguo generador de código se descuidó demasiado el aspecto visual de la herramienta, ofreciendo una interfaz pobre y poco intuitiva, de forma que un programador sin experiencia en el uso de la herramienta precisaba de explicaciones por parte de los jefes de proyecto acerca de su funcionamiento.

Genereitor persigue la facilidad de uso, proveyendo a la interfaz de títulos y literales lo más claros y concisos posible, y mostrando mediante *tooltips*[1] que clarifiquen la función que ofrecen las opciones que no resulten triviales en la presentación.

[1]Letreros que aparecen al situar el cursor sobre una opción determinada

2.1.3. Tratamiento de errores y excepciones

Otro defecto del que adolece el generador antiguo es el tratamiento de excepciones y errores en su funcionamiento. Cuando existe algún problema, no se informa al usuario de qué es lo que falla o cual puede ser la causa del problema. Este fallo se aprecia acusadamente en caso de errores de conexión a la base de datos, donde simplemente se vuelca en pantalla el contenido de la excepción generada, que generalmente suele ser poco aclarativa.

En Genereitor se ha implementa un sistema de notificación de errores, mediante una caja de mensajes en la parte superior de la interfaz, de forma que en caso de ocurrir cualquier error, el usuario sea informado de la causa de una forma clara y concisa, de manera que sea fácilmente identificable.

2.1.4. Restricciones

En el diseño de la base de datos siempre se tienen en cuenta diversas restricciones que deben cumplir las entidades que se van a almacenar, tales como campos obligatorios, longitud de cadenas de caracteres o rango máximo de campos numéricos. Pero es posible que interese ampliar estas restricciones o añadir nuevas en ciertos casos, sin modificar la base de datos, sino a nivel de aplicación.

El generador antiguo no era capaz de aplicar restricciones al manejo de la base de datos[2], tarea que de ser necesaria, tenía que ser implementada a mano.

Se ha decidido automatizar la adición de estas restricciones, de manera que se puedan modificar longitud máxima de campos en los formularios de la aplicación a generar, así como añadir la obligatoriedad de contener valores no nulos a cualquier campo, aunque el diseño de la base de datos sí lo permita.

2.1.5. Soporte para librerías propias

En la empresa se han desarrollado a lo largo del tiempo una serie de librerías y módulos de uso común, que proporcionan funciones y herramientas utilizadas por la mayoría de los proyectos que se desarrollan, tales como gestión de tablas maestras, gestión de usuarios, perfiles y grupos, gestión de contenidos, etc. –más adelante se explicarán en detalle las funciones de estos módulos–. Evidentemente es necesario adaptar el código para el uso de estas librerías, tarea que actualmente se hace a mano.

En el nuevo generador se permitirá elegir en la fase de creación de la estructura del proyecto qué módulos van a ser necesarios para la aplicación, de manera que quede disponible todo el abanico de opciones que ofrece cada una de ellos. También se modificarán los scripts de compilación y despliegue para compilar automáticamente estos módulos a la vez que el proyecto a desarrollar.

[2]Las restricciones que se pueden aplicar serán siempre a nivel de la aplicación a generar, **nunca** se modificarán las de la propia base de datos desde un generador automático.

2.1.6. Accesibilidad

Las pocas partes de la interfaz web que proporciona el generador antiguo basan parte de su funcionamiento en JavaScript. Ahora se desea que las aplicaciones desarrolladas sean accesibles, por lo que el uso de JavaScript está vetado. Por esto, Genereitor ofrecerá alternativas accesibles en forma de comentarios en el código generado, dejando también las funciones JavaScript para que el programador elija la que más convenga según los requisitos del cliente.

2.1.7. Scripts de compilación y despliegue

Una de las tareas más tediosas a la hora de comenzar la implementación de una aplicación es confeccionar los scripts de compilación. puesto que son de elaboración muy delicada.

En el desarrollo de proyectos J2EE utilizaremos para esta tarea la herramienta ant que, al ser multiplataforma, proporciona ventajas sobre otras herramientas similares tales como Makefile[3]. Con Genereitor se ofrecerán automáticamente estos scripts, personalizados de acuerdo a las características del proyecto, módulos y librerías utilizados y aplicaciones web que contiene.

2.1.8. Estructura de proyecto obsoleta

La función de generación de estructura de proyecto (esqueleto) del generador antiguo devuelve un paquete cuya organización no se corresponde con los requisitos actuales de desarrollo de aplicaciones J2EE, por lo que el programador ha de reorganizar los ficheros y directorios a mano para adaptarlos al esquema válido. Dicho esquema válido es el aconsejado en las Java BluePrints[2].

2.1.9. Integración con módulos de la empresa

En la empresa se han desarrollado a lo largo del tiempo una serie de módulos que ofrecen diversas funcionalidades a las aplicaciones desarrolladas, tales como la gestión de tablas maestras, gestión de contenidos, soporte para usuarios, grupos y perfiles, etc. El generador antiguo no ofrece soporte para estos módulos, pero Genereitor contemplará su integración opcional en las aplicaciones desarrolladas con él, y en caso de seleccionarse se crearán los ficheros específicos de dichos módulos, de manera que las aplicaciones puedan disponer de dichas funcionalidades.

2.1.10. Funcionalidades descartadas.

Con el paso del tiempo, hay tecnologías que quedan obsoletas y son sustituídas por otras más eficientes, o simplemente se decide en un momento dado apostar por alternativas que proporcionen más ventajas respecto a las actuales.

[3]En C.3 en la página 155 se detallan las características de la herramienta ant.

En generador antiguo ofrece soporte para Struts, que ya no se utiliza en la empresa y por lo tanto será descartada. También utiliza plantillas JSP para generar los diversos ficheros de código fuente, mientras que Genereitor utilizará plantillas XSL[4].

[4]En C.4 en la página 158 se detallan las características de XML/XSLT y las ventajas que tiene frente a otros sistemas.

Parte II

Estructura lógica de Genereitor

Capítulo 3

Modelo de 3 capas

Antes de comenzar a analizar las diferentes arquitecturas disponibles, conviene realizar una pequeña aclaración.

No se deben confundir los términos *capa* y nivel:

- Las capas hacen referencia a la estructura lógica de la aplicación, que repercutirá, entre otras cosas, en su modularidad y escalabilidad.
- Los niveles por contra, señalan la estructura física sobre la que se despliega la aplicación, es decir, *en cuantas máquinas corre*.

Dependiendo de la envergadura del proyecto, podemos encontrarnos aplicaciones pequeñas que estén implementadas con 3 capas, pero sólo 2 niveles, por ejemplo en la figura 3.1 en la página siguiente se muestra la distribución física de una aplicación de 3 capas, cuyas capas de acceso a datos y lógica de negocio corren bajo la misma máquina.

También se puede dar el caso en que una misma máquina soporte la capa de lógica de negocio y la capa cliente, o que todas las capas se alojen en el mismo servidor.

Por otro lado, no es raro que, en aplicaciones de gran envergadura, los servidores de bases de datos se implementen en clústeres, es decir, varias máquinas que, por computación distribuída, ofrecen la misma función y se reparten la carga de trabajo y la capacidad de almacenamiento.

En caso de Genereitor, se trata de una aplicación *de 3 capas y 3 niveles*, ya que tanto la lógica de negocio como el cliente se encontrarán en máquinas separadas. El caso «especial» de Genereitor es que no trabajará sobre una base de datos en particular, sino que está diseñado para utilizar diversos sistemas de bases de datos, que usualmente se alojarán en servidores diferentes, por lo que incluso se podría considerar como una aplicación de 4, 5 o incluso más niveles. En la figura 3.2 en la página siguiente se ilustra este ejemplo.

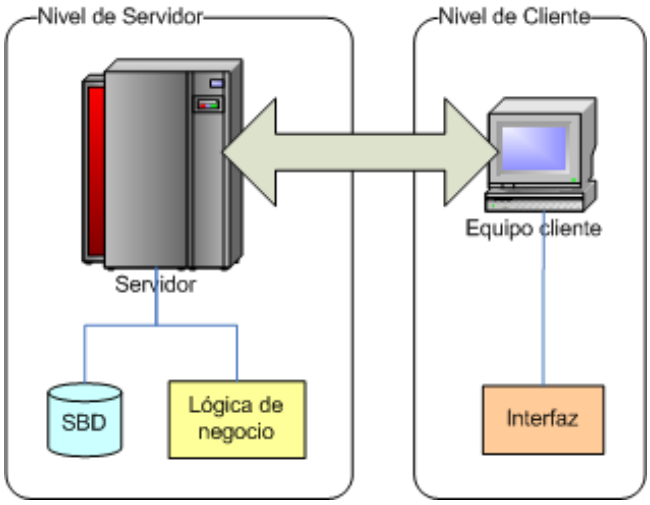

Figura 3.1: Ejemplo de aplicación de 3 capas y 2 niveles

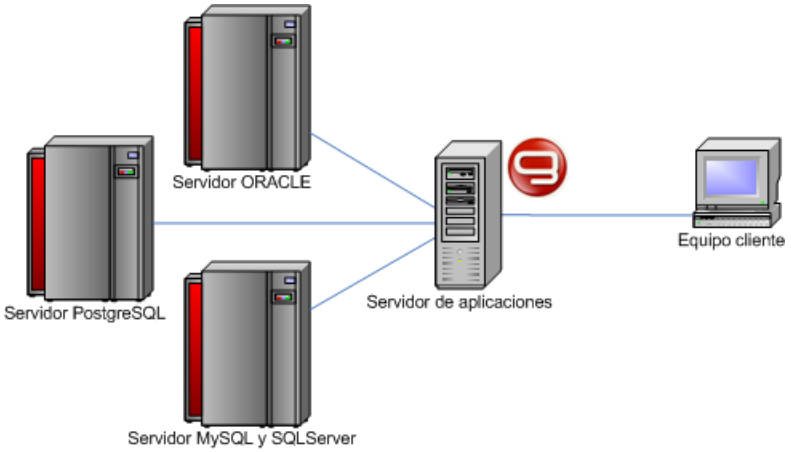

Figura 3.2: Estructura física de Genereitor.

3.1. Arquitecturas antiguas

Antes de analizar el modelo de arquitectura utilizado para la implementación de Genereitor se dará un vistazo a las características de las arquitecturas predecesoras, comentando sus ventajas e inconvenientes.

3.1.1. Aplicaciones autónomas

Una aplicación autónoma es aquella cuya arquitectura está compuesta por una sola capa, que aglutina la interfaz de usuario, la lógica de tratamiento

de datos y el almacenamiento de éstos. Las características de este modelo de aplicaciones son las siguientes:

- Su ejecución no depende de otras aplicaciones, aunque puede existir integración entre varias aplicaciones.
- Se encuentra almacenada en el disco duro del usuario.
- Su interfaz gráfica está formada por ventanas de aplicación.
- Cuando el usuario cierra la ventana, la aplicación finaliza su ejecución.
- No tienen restricciones de seguridad, pudiendo la aplicación acceder al disco duro del usuario y establecer conexiones de red.

Este diseño arcaico conlleva gran cantidad de inconvenientes:

- En cada sistema operativo se ejecuta de forma diferente, por lo que no hay compatibilidad multiplataforma.
- Problemas de integración en sistemas software complejos, tales como los sistemas de gestión de una empresa.
- Problemas de integración en sistemas de información consistentes en varias aplicaciones.
- Problemas de escalabilidad, tanto a nivel de almacenamiento como de adición de nuevas funciones.

Otro planteamiento de esta estructura monocapa es aquel consistente en un *mainframe*, un gran ordenador central que soporta todo el peso de la aplicación, y una serie de *terminales tontos*, máquinas que no realizan ningún proceso y que simplemente tienen la tarea de servir de presentación de la interfaz al usuario. Este planteamiento se representa en la figura 3.3 en la página siguiente.

Aún así, tanto la lógica de la aplicación, el acceso a datos y la presetación de la información estaba completamente implementada en un sólo bloque monolítico de software. Cualquier modificación sobre la aplicación requería modificar la totalidad de este único bloque.

Un avance sobre este modelo fue realizado a partir de bases de datos basadas en servidores de archivos. En este caso, la base de datos consiste en uno o más archivos reconocibles por el sistema operativo. En esta arquitectura, el programa que permite el acceso y administración de la base de datos debe estar muy estrechamente unido a la aplicación cliente.

3.1.2. Aplicaciones con conexión a BD: 2 capas

Un avance más a la arquitectura anterior consiste en dividir los sistemas de una sola capa en dos capas bien diferenciadas. Es lo que se conoce como *arquitectura cliente-servidor*.

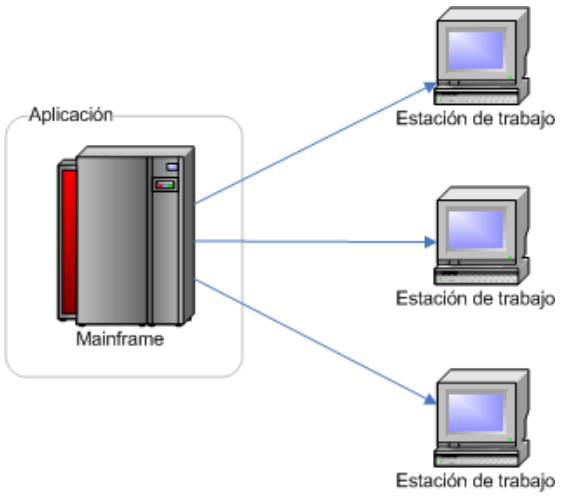

Figura 3.3: Esquema de la arquitectura monocapa con *mainframe* y *terminales tontas*

Estas aplicaciones están compuestas por dos capas[1], tal como se muestra en la figura 3.4:

Front-end: es la capa donde el usuario interactúa con su máquina y que, generalmente, aglutina toda la lógica de negocio.

Back-end: es la capa de acceso a datos, cuya función la realiza un servidor de bases de datos y reside en un servidor central bajo un entorno controlado.

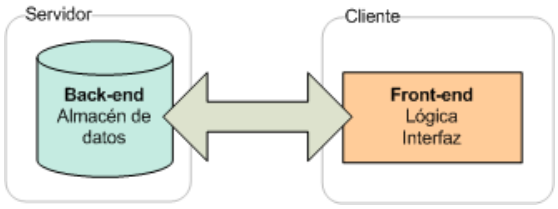

Figura 3.4: Esquema de la arquitectura de 2 capas

Uno de los problemas en este tipod e arquitecturas es la dificultad de manipular los cambios en la capa que interactúa con el cliente. En estos casos, varias estaciones de trabajo clientes necesitarán ser actualizadas con una nueva versión de la aplicación del cliente simultáneamente al cambio en la base de datos. Ésta

[1]Es importante no confundir esta arquitectura con la del mainframe monocapa. En la arquitectura de 2 capas, la lógica de la aplicación reside en la capa cliente (front-end), que se ejecuta en cada máquina cliente. En la estructura anteriormente mencionada, las estaciones de trabajo son únicamente dispositivos de entrada (teclado, ratón) y salida (pantalla) de información, no realizando éstos ninguna tarea relacionada con la aplicación, más que la de meros interfaces.

generalmente no es una tarea sencilla, sobre todo si las aplicaciones cliente están geográficamente dispersas.

Otro problema es la dificultad de compartir procesos comunes. Tras largas horas de trabajo frente a la máquina para lograr un proceso en particular, este código es difícilmente reutilizable en otras aplicaciones.

Un problema más es la seguridad. Ésta puede ser establecida en cualquera de las dos capas, pero cada una tiene sus limitaciones. La primera solución consiste en dar privilegios a cada uno de los objetos que componen la base de datos y a los usuarios. Sin embargo, las corporaciones no requieren sólo asegurar *qué datos* pueden ser actualizados o accedidos, sino *de qué manera*. En cuanto al segundo punto, que es el más usado, aunque el usuario puede acceder a la base de datos con su identificación, tiene dos problemas:

- Dado que ninguno de los objetos en la base de datos es seguro, cualquier usuario puede tener acceso total a la misma con alguna aplicación de front-end.
- La implantación de la seguridad deberá ser desarrollada, probada y mantenida en absolutamente toda la red, sin importar dónde se encuentren las estaciones cliente.

Otros inconvenientes son:

- Los servidores de bases de datos no proporcionan un lenguaje de programación *completo*[2], con control de flujo, y los procedimientos almacenados[3], aunque ayudan, no son la solución.
- Los datos no están encapsulados, por lo que sigue siendo necesario que el programador de las aplicaciones de los clientes realice tareas de control de integridad de los datos.
- No resulta fácil realizar cambios en la estructura de una base de datos de la que dependen varias aplicaciones.
- A medida que el negocio crece y el número de usuarios simultáneos del sistema aumenta, se complican las opciones de escalabilidad del sistema. Aplicaciones de 2 capas funcionan perfectamente en entornos pequeños, pero no son capaces de acompañar un gran crecimiento del negocio.
- Las actualizaciones de la aplicación cliente se han de distribuir entre todos los usuarios, que deberán realizar una instalación de la nueva versión del producto en sus máquinas. En sistemas con muchos usuarios, esta tarea será costosa a nivel tanto económico como temporal.

[2]Para suplir esta carencia, los diversos sistemas de bases de datos están desarrollando sus propios lenguajes procedurales, de modo que amplían la funcionalidad de los procedimientos almacenados, tales como PL/SQL de Oracle o PL/PgSQL de Postgres.

[3]Los procedimientos almacenados son programas que se almacenan físicamente en una base de datos. Ofrecen la ventaja de que son ejecutados directamente en el motor de bases de datos, y como tal posee acceso directo a los datos que necesita manipular y sólo necesita enviar los resultados de regreso a la capa superior, deshaciéndose de la sobrecarga resultante de transferir grandes cantidades de datos.

- El implementar la lógica de acceso a datos en el propio cliente implica que un usuario puede descifrar su funcionamiento, y por tanto su implementación, lo que posibilita el aprovechamiento de fallos de seguridad para llevar a cabo acciones que no están previstas.

Por contra, algunas de las ventajas que ofrece este sistema son:

- Cuando un servidor de bases de datos procesa una consulta, la eficiencia en la devolución de la respuesta a esta petición dependerá de la máquina donde se encuentra alojado el servidor, y no de la del cliente, que únicamente recibe el resultado.
- El servidor de datos devuelve sólo la informaciñón solicitada a través de la red, de tal modo que el tráfico de la misma resulta sustancialmente reducido. Esto permite crear aplicaciones que acceden a grandes cantidades de datos utilizando anchos de banda no excesivamente amplios.
- Un servidor de bases de datos puede asegurar más eficazmente la integridad y consistencia de los datos que los sistemas utilizados anteriormente, tales como servidores de archivos.

3.2. Arquitectura de 3 capas.

La estrategia tradicional de utilizar aplicaciones compactas causa gran cantidad de problemas de integración en sistemas software complejos como pueden ser los sistemas de gestión de una empresa o los sistemas de información integrados consistentes en más de una aplicación. Estas aplicaciones, como ya hemos visto, suelen encontrarse con importantes problemas de escalabilidad, disponibilidad, seguridad e integración.

El paso siguiente a la arquitectura de 2 capas fue la aparición entre la capa de interfaz (presentación) y la de acceso a datos, de una tercera capa de reglas o lógica de negocio, que es la que realmente implementa las funciones de la aplicación y debe obviar tanto la estructura de los datos como su ubicación. El cliente «pesado» que en la arquitectura de dos capas aglutina la interfaz junto con la lógica de la aplicación se divide en un cliente «ligero» y la lógica de la aplicación se traslada completamente a un servidor. Por ejemplo, en una aplicación web el cliente es un navegador que muestra las páginas enviadas por el servidor que administra la lógica de negocio, las cuales permiten el ingreso o la consulta de los datos.

3.2.1. Funciones de cada capa

En la figura 3.5 en la página siguiente se ilustran de manera general los componentes de la arquitectura de 3 capas.

Figura 3.5: Esquema de la arquitectura de 3 capas

3.2.1.1. Acceso a datos

Es la capa de nivel más bajo. Sus funciones incluyen el almacenamiento, la actualización y la consulta de todos los datos contenidos en el sistema. En la práctica, esta capa es esencialmente un servidor de bases de datos, aunque podría ser cualquier otra fuente de información. Gracias a esta división, es posible agregar soporte para una nueva base de datos en un período de tiempo relativamente corto. La capa de datos puede estar en el mismo servidor que las de lógica de negocio y presentación, en un servidor independiente o incluso estar distribuida entre un conjunto de servidores, dependiendo de la magnitud de la aplicación.

3.2.1.2. Capa de aplicación

El comportamiento de la aplicación es definido por los componentes que modelan la lógica de negocio. Estos componentes reciben las acciones a realizar a través de la capa de presentación, y lleva a cabo las tareas necesarias utilizando la capa de datos para manipular la información del sistema. Tener la lógica de negocio separada del resto del sistema también permite una integración más sencilla y eficaz con sistemas externos, ya que la misma lógica utilizada por la capa de presentación puede ser accedida desde procesos automáticos que intercambian información con los mismos.

3.2.1.3. Capa de presentación o capa cliente

La capa de presentación representa la parte del sistema con la que interactúa el usuario, la *interfaz*. En una aplicación web, un navegador puede utilizarse como cliente del sistema, pero ésta no es la única posibilidad, también puede generarse una aplicación que cumpla las funciones de cliente «ligero» para interactuar con el usuario.

3.2.2. Ventajas de la arquitectura de 3 capas

La arquitectura de 3 capas tiene todas las ventajas de los sistemas cliente/-servidor, además de las que de por sí tienen los sistemas diseñados de forma modular. Pero también han conseguido mejorar muchos de los aspectos que han resultado difíciles de solucionar en la arquitectura de 2 capas:

Permite la reutilización: La aplicación está formada por una serie de componentes que se comunican entre sí a través de interfaces y que cooperan para lograr el comportamiento deseado. Esto permite no solamente que estos componentes puedan ser fácilmente reemplazados por otros, por ejemplo en caso de necesitarse mayor funcionalidad, sino también que los mismos puedan ser utilizados para otras aplicaciones.

Acompaña el crecimiento: Cada uno de los componentes de la aplicación pueden colocarse en el mismo equipo o distribuirse a través de una red. De esta manera, proyectos de gran envergadura pueden dividirse en pequeños proyectos más simples y manejables, que se pueden implementar en forma progresiva, agregando nuevos servicios según la medida de crecimiento de la organización.

Uso eficiente del hardware: Debido a que los componentes pueden ser distribuidos a través de toda la red, se puede hacer un uso más eficiente de los recursos de hardware. En vez de necesitarse grandes servidores que contengan la lógica de negocios y los datos, es posible distribuirlos en varias máquinas más pequeñas, económicas y de fácil reemplazo en caso de fallo.

Mínima inversión inicial: Generalmente, un cambio en el sistema de gestión traía asociada una inversión importante en la actualización de hardware en los clientes, debido a nuevas necesidades de cómputo de las aplicaciones «pesadas». Los clientes «ligeros» de esta nueva modalidad permiten manterner el equipamiento actual o adquirir uno de muy bajo coste y actualizar, sólo en caso de ser necesario, la tecnología de los servidores.

Distintas presentaciones: Debido a que separa la presentación de la lógica de negocio, es mucho más sencillo realizar tantas presentaciones diferentes como dispositivos con capacidades e interfaces se tenga (PC, PDA, teléfonos móviles, etc.).

Encapsulación de los datos: Debido a que las aplicaciones cliente se comunican con los datos a través de peticiones que los servidores responden ocultando y encapsulando los detalles de la lógica de la aplicación, obtenemos un nivel de abstracción que permite un acceso a los datos consistente, seguro y auditable. Con esto se pretende lograr que si hay cambios en la capa de datos, la capa de negocios se haga cargo de administrar tales cambios de forma transparente de forma que el cliente permanezca ajeno a esos cambios.

Mejor calidad final de las aplicaciones: Como las aplicaciones son construidas en unidades separadas, éstas pueden ser probadas independientemente y con mucho más detalle, lo que conduce a obtener un producto mucho más sólido y fiable.

Capítulo 4

Arquitectura J2EE

J2EE es una tecnología que pretende simplificar el diseño y la implementación de aplicaciones empresariales.

Una **aplicación empresarial** es una aplicación que probablemente disponga de aplicaciones y bases de datos ya existentes, que se quieran seguir utilizando durante la migración a un nuevo conjunto de herramientas que exploten las posibilidades de Internet, comercio electrónico y otras nuevas tecnologías.

Las razones principales de la evolución de las aplicaciones empresariales son:

- Necesidad de aprovechar las nuevas tecnologías, especialmente los avances en sistemas web.
- Necesidad de manejar detalles a bajo nivel, propias e inherentes a toda aplicación empresarial, tales como seguridad, proceso de transacciones y tareas multihilo.
- Evolución y popularidad de conceptos ampliamente aceptados, tales como la arquitectura multicapa y el desarrollo de software basada en componentes.

Muchos entornos de desarrollo (*frameworks*) de aplicaciones empresariales han aparecido a partir de estas necesidades. Algunos de los ejemplos más conocidos son Java2 Platform Enterprise Edition, J2EE, de Sun Microsystems, Distributed Internet Applications Architecture, DNA, de Microsoft y Common Object Request Broker Architecture, CORBA, de Object Management Group (OMG).

4.1. *¿Por qué* J2EE*?*

Tal vez J2EE no sea la tecnología definitiva para el desarrollo de aplicaciones empresariales, no obstante ofrece una serie de ventajas que la han hecho posicionarse como una de las alternativas más utilizadas:

4.1.1. Ahorro de trabajo

Una aplicación empresarial necesita implementar servicios realmente complejos para ser completamente funcional y eficiente. Ejemplos de estos servicios son el manejo de estados y transacciones, *pooling* de recursos o gestión de tareas multihilo. La arquitectura J2EE separa estos servicios de bajo nivel de la lógica de la aplicación, puesto que están implementados en el propio servidor de aplicaciones. De este modo, no es necesario implementarlos para cada proyecto en caso de ser necesarios.

4.1.2. Documentación

J2EE es desarrollado por un consorcio formado por grandes compañías, el Java Community Process[1], lo que garantiza que su implementación está documentada de forma completa y normalizada.

Asimismo, las APIs utilizadas también cuentan con documentación completa y detallada de todas sus funciones. Esta documentación puede ser consultada directamente desde la página de Sun, existiendo documentación para todas las versiones de la especificación.

4.1.3. Estándar y confiable

El uso de una arquitectura madura y que se ha convertido en estándar *de facto* implica la posibilidad de desarrollar aplicaciones confiables, lo que se traduce en menor gasto de mantenimiento y garantiza su longevidad y escalabilidad.

Esta especificación es considerada informalmente como un estándar, debido a que los suministradores deben cumplir ciertos requisitos de conformidad para declarar que sus productos son *conformes a J2EE*, no obstante no se trata de un estandar ISO o ECMA.

4.1.4. Flexible

Las aplicaciones desarrolladas con J2EE gozan de gran flexibilidad. Es posible desplegarlas en cualquier servidor de aplicaciones haciendo sólamente unos pocos cambios en los ficheros de configuración. De hecho, aunque Genereitor está desarrollado para trabajar sobre Tomcat, se ha conseguido desplegar sobre OC4J y JBoss sin problemas, haciendo mínimos cambios en sus ficheros de configuración.

4.2. Plataforma J2EE

La plataforma J2EE utiliza un modelo de aplicaciones distribuído y multicapa. La lógica de la aplicación está dividida en componentes, y cada componente

[1] La especificación de J2EE se puede consultar en http://www.jcp.org/en/jsr/detail?id=244.

está instalado en una máquina específica, cuyas características irán acordes con los requisitos de los componentes que se alojan en ella[2]. En la figura 4.1 se muestra un diagrama básico de la arquitectura J2EE.

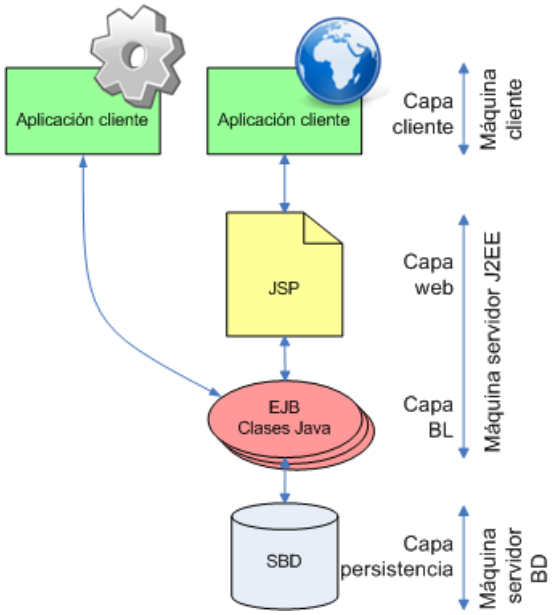

Figura 4.1: Diagrama de la arquitectura J2EE

4.3. *Model-View-Controller*

El concepto de la arquitectura Modelo-Vista-Controlador se basa en separar el modelo de datos de la aplicación de su representación de cara al usuario, y de la interacción de éste con la aplicación, mediante la división del sistema en tres partes fundamentales:

Modelo: contiene la lógica de negocio de la aplicación.

Vista: muestra al usuario la información que éste solicita.

Controlador: recibe e interpreta la interacción del usuario, actuando sobre modelo y vista de manera adecuada para provocar cambios de estado en la representación interna de los datos, así como en su visualización.

Un esquema de estas tres capas se presenta en la figura 4.2 en la página siguiente.

[2]Aunque la arquitectura esté pensada para que las diversas capas estén soportadas por máquinas diferentes, es perfectamente posible, en el caso de aplicaciones pequeñas, que todos los servidores residan en la misma máquina. Asímismo, para aplicaciones de gran envergadura, también es posible que una capa sea soportada por varios servidores, utilizando computación distribuida.

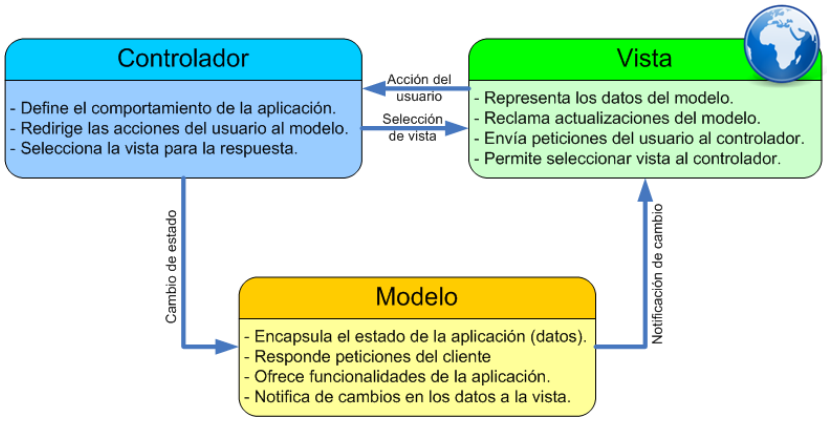

Figura 4.2: Diagrama del MVC.

Las consecuencias del uso de MVC en el diseño de una aplicación son:

Componentes del Modelo reutilizables. La separación entre modelo y vista permite implementar múltiples vistas que utilicen el mismo modelo. A consecuencia de ésto, los componentes del modelo de una aplicación empresarial son más fáciles de implmenetar, probar y mantener, puesto que el modelo se basa en estos componentes.

Fácil soporte para nuevos tipos de clientes. Para incluir una nueva vista en una aplicación (móviles, pdas, etc) sólo es necesario implementar dicha vista y la lógica de control del modelo, e incorporarlos a la aplicación. Dicha vista aprovechará el modelo existente para funcionar.

Incrementa la complejidad del diseño. Este esquema incorpora algunas clases adicionales, debido a la separación de vista, modelo y controlador, por lo que el diseño aumenta de tamaño y, en aplicaciones pequeñas, puede ser engorroso mantener tal separación.

Esta arquitectura ha demostrado ser muy apropiada para las aplicaciones web, y se adapta extraordinariamente a las tecnologías proporcionadas por la plataforma J2EE. Así, para implementar una arquitectura MVC sobre J2EE, los componentes de la misma serían:

4.3.1. Modelo

El modelo representa los datos de la aplicación y la lógica de negocio que gobierna el acceso a dichos datos. Estará implementado por un conjunto de clases Java. Para este cometido existen dos alternativas de implementación:

4.3.1.1. Enterprise JavaBeans

Enterprise JavaBeans es una arquitectura de componentes distribuído del lado del servidor para la construcción modular de aplicaciones empresariales.

La especificación EJB es una de las muchas APIs de J2EE[3], que intenta proveer un método estándar para implementar la lógica de negocio que se suele encontrar en aplicaciones empresariales. Fue concebida para manejar características tales como persistencia, integridad transaccional y seguridad de forma estándar, liberando a los programadores para concentrarse en las tareas específicas de cada aplicación.

La especificación EJB provee:

- Persistencia.
- Procesado transaccional[4].
- Control de concurrencia.
- Eventos, usando Java Message Service.
- Seguridad.
- Despliegue de componentes de software en el servidor de aplicaciones.

4.3.1.2. Plain Old Java Objects

Los POJOs son clases Java simples, que no dependen de un *framework* en especial. Es una apuesta por la simplicidad en proyectos de pequeña envergadura, en los que es posible prescindir de los complejos frameworks y la arquitectura que implica el uso de EJBs.

Las características de un POJO son:

- Es serializable.
- Tiene constructor sin argumentos.
- Permite el acceso a sus propiedades mediante métodos `getPropiedad()` y `setPropiedad()`.

Genereitor implementa su capa modelo con POJOs, aunque los proyectos generados con él incluyen la posibilidad de implementarla mediante EJB.

[3] La especificación de EJB 2.0 se puede descargar en http://www.jcp.org/en/jsr/detail?id=19/.

[4] La gestión transaccional es un método consistente en descomponer los procesos en operaciones atómicas e indivisibles llamadas transacciones. Cada transacción debe finalizar de forma correcta o incorrecta como una unidad completa, no puede acabar en un estado intermedio. También establece características como *rollback*, que permite, en caso de fallo, deshacer la operación para restaurar su estado inicial e informar del fallo.

4.3.2. Vista

La vista proporciona una serie de páginas web, creadas dinámicamente, al cliente, que las recibirá como simples páginas HTML que interpretará su navegador.

Existen múltiples frameworks que generan estas páginas web a partir de distintos formatos, siendo el más extendido JSP (*Java Servlets Page*[5]).

JSP permite desarrollar páginas HTML, pero incluye la posibilidad de utilizar en éstas código Java, lo que permite a personas con conocimientos en HTML desarrollar interfaces de una forma rápida y sencilla.

Alternativas a JSP son ASP, PHP y Phyton, no obstante la primera es la que ofrece una mejor integración con J2EE.

4.3.3. Controlador

El controlador traduce las interacciones del usuario con la vista a *acciones* que son interpretables por el modelo. En un cliente gráfico autónomo (*stand-alone*), las interacciones del usuario pueden ser pulsaciones de botones o selecciones en menús, mientras que en una aplicación web, estas interacciones están representadas por peticiones HTTP (GET y POST). Las acciones implementadas por el modelo pueden ser la ejecución de procesos o el cambio de estado del modelo. Basado en las interacciones del usuario y las respuestas de las acciones del modelo, el controlador responde seleccionando una vista apropiada a la petición del usuario.

En la plataforma J2EE el controlador se desarrolla mediante servlets.

Un servlet es un objeto que se ejecuta en un servidor J2EE (como Oracle Container for Java, OC4J), o un contenedor (como Tomcat)[6], y que ha sido diseñado eespecialmente para ofrecer contenido dinámico desde un servidor web, generalmente HTML.

[5]La especificación de JSP se puede consultar en http://www.jcp.org/en/jsr/detail?id=53.

[6]La diferencia entre un servidor de aplicaciones y un contenedor de servlets es que el primero extiende esta funcionalidad, proporcionando contenedor para objetos más avanzados, tales como EJB. Como en Genereitor no se utiliza EJB, se ha desarrollado para desplegar sobre Tomcat. No obstante, se puede desplegar sobre OC4J simplemente cambiando algunos parámetros de configuración.

Parte III

Funcionalidades de Genereitor

En esta parte de la memoria se van a describir las capacidades de Genereitor como herramienta de ayuda a la implementación de aplicaciones web para entornos empresariales.

Genereitor se divide en tres grandes bloques funcionales, correspondientes a tres funciones que, aunque son diferentes, se complementan para conseguir formar un *todo* que será la aplicación. Esta aplicación en realidad sólo será un esbozo del resultado final, pero sentará las bases para el desarrollo posterior por parte de los programadores, ahorrando una cantidad sustancial de tiempo.

Para explicar los diferentes apartados se hará referencia a una hipotética aplicación «taller», cuyo esquema de la base de datos se refleja en la figura 4.3.

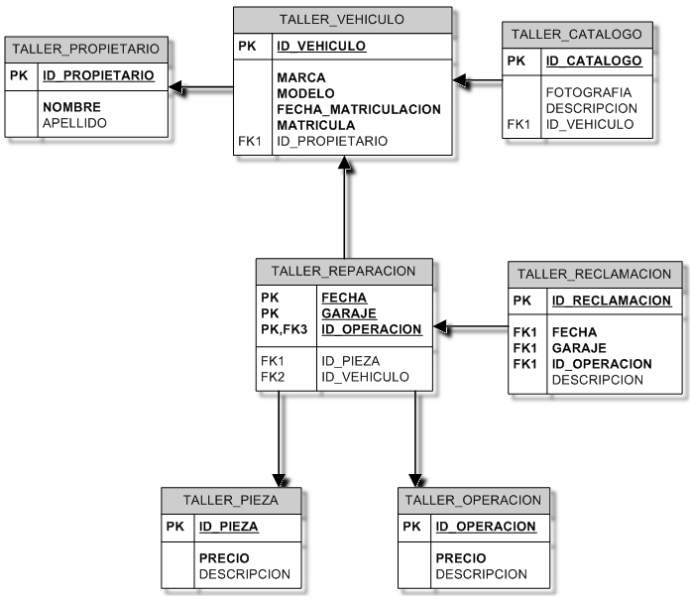

Figura 4.3: Esquema de la base de datos de la aplicación de ejemplo «taller».

Capítulo 5

Generación de esqueleto

La generación del esqueleto es la primera fase de la implementación de una aplicación web.

Anteriormente se habrá realizado el **diseño** de la aplicación, teniendo en cuenta, según los requisitos del cliente, la estructura que tomará el proyecto y las tecnologías a utilizar. Este diseño quedará reflejado con detalle en un *documento de análisis y diseño* (ver A en la página 116 para consultar el de Genereitor), que servirá de guía a lo largo de todo el proceso de implementación.

Una vez han quedado claramente reflejados todos los requisitos y características que ha de tener el proyecto en el documento de análisis y diseño, se procede a diseñar e implementar la base de datos sobre la que se apoyará la aplicación. Este paso no es común a todas las aplicaciones, puesto que es común que el cliente proporcione ya una base de datos existente (en caso de migraciones o actualizaciones de aplicación).

Tras esto, se comienza la implementación del proyecto, generando el esqueleto.

El esqueleto de una aplicación es la estructura de directorios sobre la que se asienta, y que permite tener separados y bien identificados los componentes de la misma según su función y la capa a la que pertenecen.

Junto a la estructura de directorios se generarán ficheros que implementarán funciones relativas a cada capa de la aplicación, y que serán explicados con detalle más adelante. Algunos de estos ficheros se generarán como clases vacías, con la cabecera pero sin métodos. Estos métodos serán proporcionados con la *generación de código*, operación que se realizará para cada entidad de la base de datos que quiera ser tenida en cuenta en la aplicación, y posteriormente serán añadidos a estos ficheros «incompletos» con una herramienta auxiliar (Combineitor, ver apartado B en la página 145).

5.1. Funciones de la interfaz

5.1.1. Fase 1. Generación de esqueleto

La única pantalla de generación de esqueleto contiene tres bloques que solicitan información al programador para adecuar el esqueleto generado a las necesidades del proyecto.

5.1.1.1. Bloque de datos generales

En este primer bloque se solicitan los datos generales necesarios para generación del esqueleto de una aplicación. Tales datos son:

Sistema de bases de datos. El programador elegirá de entre las opciones disponibles (Oracle 9, PostgreSQL, MySQL, HSQLDB o SQLServer[1]) el sistema de bases de datos que se ha elegido en el proceso de análisis para soportar la base de datos. Esta opción afectará tanto a los *datasources* como a la forma de acceder a los datos en la base de datos.

Servidor de aplicaciones. Según el diseño de la aplicación, se elegirá entre OC4J o Tomcat[2].

Nombre de la aplicación. El nombre de la aplicación.

Nombre del paquete base de la aplicación. El nombre del paquete base de la aplicación[3].

En este bloque también se ofrece un pulsador con el literal «*Default*», cuya función es autocompletar el siguiente bloque con los datos de conexión de las bases de datos de desarrollo que hay en la factoría de software de la empresa, según el servidor de bases de datos seleccionado. Con varias pulsaciones sobre este literal se rota de forma cíclica entre las diferentes conexiones para un mismo sistema de bases de datos, puesto que se da el caso que para algunos sistemas hay varias bases de datos de desarrollo.

5.1.1.2. Bloque de datos de conexión

Aquí se recogen los datos de conexión de la base de datos contra la que va a trabajar la aplicación a desarrollar:

- Host.

[1] El soporte para otros sistemas de bases de datos se contempla como trabajo futuro.

[2] El soporte para otros servidores de aplicaciones o contenedores de EJB se contepla como trabajo futuro.

[3] Es requisito que el nombre de la aplicación siempre sea el último elemento del nombre del paquete base. Así, la aplicación taller tendrá su paquete base llamado com.comex.**taller**. Es por esto que en la casilla de «nombre de paquete base» sólo hay que introducir los primeros elementos del nombre (com.comex), ya que el último se completa automáticamente a partir del contenido de la casilla «nombre de la aplicación».

- Puerto.
- SID, o nombre de la base de datos.
- Usuario.
- Contraseña.

Aunque para la generación del esqueleto realmente no es necesaria una conexión real a la base de datos, éstos se requieren para configurar correctamente los *dataosurces*.

Para comprobar que los datos introducidos son correctos, y así evitar errores tipográficos, se proporciona un botón con el literal «*Test*» que lanza una conexión al servidor cuyos datos han sido introducidos, informando al programador de si ésta ha sido exitosa, o del error producido en caso contrario.

5.1.1.3. Bloque de opciones de la aplicación

En este último bloque de datos se solicitan las características particulares del proyecto a generar:

Módulos web. Una aplicación web puede estar formada por varios módulos o subaplicaciones. Por ejemplo, en el caso de un portal que tenga una parte de acceso público (por ejemplo taller-web), una para clientes (taller-clientes) y otra para administración de contenidos (taller-admin). Se ofrece una lista en la que se añadirán todos los módulos que sean necesarios para la aplicación.

Utilizar Authenticator. Permite añadir un mecanismo de seguridad a las conexiones de la aplicación, consistente en un parámetro authenticator, que confirma que el usuario que solicita la conexión es un usuario válido conforme al registro de usuarios existente en la base de datos.

Utilizar XML[4]. Integra en la aplicación el módulo de comunicación de datos mediante XML de la empresa (com.comex.xml).

Utilizar CMS. Integra en la aplicación el sistema de gestión de contenidos (com.comex.cms).

Utilizar parámetros. Hace que la aplicación generada tome una serie de parámetros que se alojarán en la base de datos, en lugar de almacenarlos en la propia aplicación.[5]

Utilizar tablas maestras. Integra en la aplicación el módulo de tablas maestras (com.comex.tablasmaestras), que genera las interfaces de gestión de

[4] La función XML todavía no está implementada completamente.

[5] Actualmente la opción *parámetros* es obligatoria, puesto que otros módulos dependen de estos parámetros. Más adelante, cuando se actualicen esos módulos y no requieran los parámetros alojados en la base de datos, esta posibilidad se ofrecerá como opcional.

tablas maestras a partir de un script que se generará a posteriori con la función «*Generar script de tablas maestras*» de Genereitor.[6]

Usuarios, perfiles y grupos[7]. Integra en la aplicación el módulo que da soporte para usuarios, grupos de usuarios y perfiles, y ofrece una lista para configurar los perfiles que se utilizarán en la aplicación.

Librerías. Ofrece una lista de las librerías comúnmente utilizadas, de manera que el programador incluya las que crea convenientes teniendo en cuenta los requisitos del diseño. Estas librerías se incluirán en el esqueleto generado, y se tendrán en cuenta en los scripts de compilación y despliegue de la aplicación para que no haya que moverlas *a mano*.

Por último, tras haber rellenado las opciones pertinentes y al pulsar el botón «*Generar*», el programa devuelve un paquete comprimido que contiene el esqueleto de la nueva aplicación.

5.2. Resultado

Tras haber generado el esqueleto, se ha obtenido un paquete que contiene la estructura del proyecto comprimida. El siguiente paso será descomprimir su contenido en el directorio de la máquina del desarrollador, que será su copia de trabajo.

De forma muy simplificada, la estructura de directorios generada es la representada en la figura 5.1 en la página siguiente.

Además de la estructura de directorios, se generan diversos archivos que a continuación se detallan:

5.2.1. Ficheros properties y descriptores de despliegue.

acciones.properties: Este fichero contiene todas las acciones que se ofrecen en la interfaz de usuario, y la correspondencia con el servlet que las implementa.

La interfaz (vista), que es una página HTML, contiene vínculos a las acciones (por ejemplo, a vehiculo.listar.do), que hacen referencia a los métodos de los servlets (*Actions*). Este fichero se encarga de interpretar las peticiones del usuario, mediante su interacción con la interfaz, e invocar al servlet correspondiente.

etiquetas.properties: Este fichero contiene los textos mostrados en la interfaz web. En lugar de escribirlos directamente en el fichero JSP, se utiliza este fichero que contiene los literales de la interfaz identificados con una etiqueta, que será la que se use al implementar la interfaz. De este modo se

[6] Actualmente la opción tablas maestras es obligatoria, puesto que otros módulos dependen de funcionalidades implementadas en com.comex.tablasmaestras. Cuando se actualicen estas dependencias y no se requiera del módulo de tablas maestras, esta posibilidad se ofrecerá como opcional.

[7] La función Usuarios todavía no está implementada completamente.

```
/
├── META-INF
├── apps
│   └── taller
│       ├── taller-ejb
│       │   ├── ejb ..... Fachada y EJB
│       │   └── exception ..... Excepciones
│       ├── taller-web ..... Módulo web «web»
│       │   ├── src
│       │   │   ├── servlet ..... Actions
│       │   │   ├── taglib ..... Tags de la parte web
│       │   │   └── util
│       │   └── web ..... JSP, CSS, imágenes
│       ├── taller-webadmin ..... Módulo web «webadmin»
│       │   └── ...
│       ├── ...
│       └── src ..... Ficheros de configuración, despliegue, data-sources...
├── components
│   ├── common ..... Módulos utilizados
│   └── taller
│       ├── dal ..... Acceso a datos
│       │   └── bean ..... Implementación de los objetos
├── lib ..... Librerías
└── setup
    ├── instalacion ..... Ayuda a la instalación
    └── scripts ..... Scripts SQL
```

Figura 5.1: Árbol de directorios simplificado de una aplicación web.

```
/
└── apps/
    └── taller/
        └── src/
            └── conf/
                └── propertyFiles/
                    ├── etiquetas.properties
                    ├── acciones.properties
                    ├── des.log4j.properties
                    ├── pre.log4j.properties
                    ├── pro.log4j.properties
                    ├── des.config.properties
                    ├── pre.config.properties
                    └── pro.config.properties
```

Figura 5.2: Ruta de los ficheros .properties.

permite el soporte para varios idiomas, o la rápida localización y corrección de errores en la escritura de los textos de la aplicación.

log4j.properties: Contiene la configuración de los logs de la aplicación. Se ofrecen tres configuraciones (para desarrollo, preproducción y producción), puesto que según la etapa en la que se encuentre el proyecto será necesaria una mayor o menor intensidad en la monitorización y auditoría de los eventos que tengan lugar.

config.properties: Contiene diversos parámetros de configuración general propios de la aplicación.

En caso de que la aplicación utilice EJBs, se proporcionarán también los archivos (incompletos, se completarán con la *generación de código*) que contendrán los descriptores de despliegue. Estos archivos indican al servidor de aplicaciones cómo han de desplegar la aplicación para ponerla en funcionamiento y el tratamiento que han de dar a los EJBs.

5.2.2. Scripts de compilación

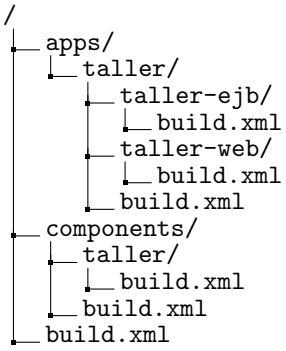

Figura 5.3: Ruta de los scripts de compilación.

Estos archivos build.xml repartidos por toda la estructura de directorios son los que indican a *ant* de qué forma se debe llevar a cabo la compilación del proyecto.

El script principal es el situado en el directorio raíz (`/build.xml`), que irá llamando según convenga a los situados en los diferentes directorios de la estructura. El motivo de la división de las tareas de compilación en varios scripts es la mejora de la modularidad, manteniendo independencia entre las diferentes partes del proyecto.

Junto a cada script build.xml se proporciona otro, build_para_envio.xml, cuya función es idéntica al anterior, pero teniendo en cuenta los parámetros de la máquina del cliente donde se va a desplegar para su entrada en producción, por lo que algunos parámetros serán diferentes.

5.2.3. Capa web - Servlets (Controlador)

En esta ruta se alojan los ficheros fuente de los servlets, tags y demás utilidades que utiliza el controlador de la capa de aplicación para generar la interfaz web y comunicarse con el modelo de la capa de lógica de negocio.

Las funciones de los archivos generados son las siguientes:

servlet/InicioAction.java: Implementa el servlet de bienvenida a la aplicación, que se encargará de formar la primera página de la interfaz.

servlet/ServletEscuchador.java: Se trata de un servlet que monitoriza las sesiones para crear y destruir las sesiones que permanezcan demasiado tiempo

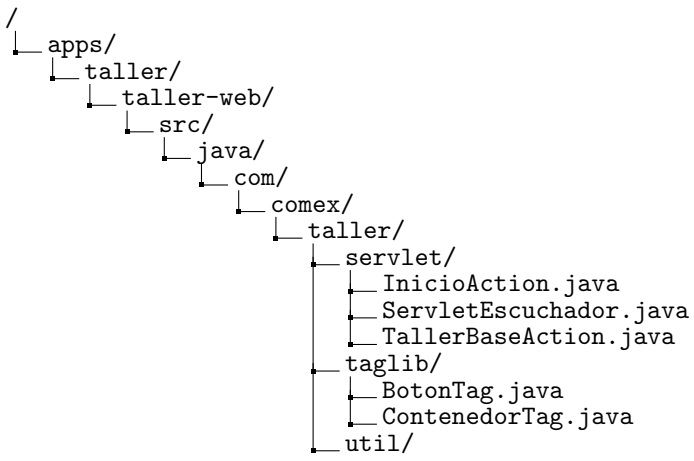

Figura 5.4: Ruta de los servlets de un módulo web.

inactivas, y por tanto se considere que han caducado. También limpiará cada cierto tiempo los atributos de dichas sesiones inactivas.

servlet/TallerBaseAction.java: Servlet base que extenderán todos los demás de la aplicación. Implementa también funciones comunes o usadas por varios servlets, con objeto de reutilizar código. También implementa los métodos para que los demás servlets sean capaces de comunicarse con la fachada que da acceso al modelo de la aplicación.

taglib/ContenedorTag.java y taglib/BotonTag.java: Implementación de los tags que se utilizarán en los JSP para dibujar el contenedor general de la interfaz y los botones. Esto supone una gran reutilización de código, consiguiendo además que todas las páginas pertenecientes al mismo módulo web tengan la misma apariencia.

Directorio util: Aquí se localizan otros tags más genéricos, que no se modifican de uno proyecto a otro, como el tag Paginador, que se encarga de agrupar los resultados de las listas en páginas de n elementos, ofreciendo enlaces de «siguiente», «anterior», etc, y también implementa la función de ordenar.

También se encuentran aquí ficheros que almacenan métodos para validar de forma accesible diversos eventos en los formularios de la interfaz, como comprobar que un NIF exista o que la fecha sea correcta, y diversas funciones de formateo de números y cadenas.

5.2.4. Capa web - JSP (Vista)

En la carpeta `taller-web/web` se alojan todos los ficheros fuente directamente relacionados con la interfaz y su aspecto (figura 5.5 en la página siguiente):

Los directorios mostrados contienen los siguientes ficheros:

WEB-INF: Contiene los descriptores de los tags que están disponibles para su uso en la interfaz, de forma que los ficheros JSP sean capaces de inter-

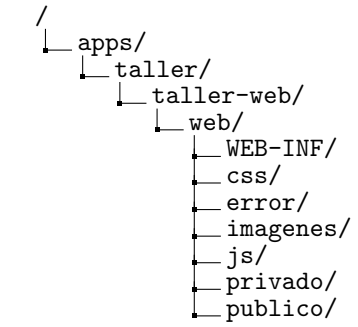

Figura 5.5: Ruta de la parte web (interfaz) de un módulo web.

pretarlos según su implementación en el directorio del apartado anterior. También contiene el fichero web.xml[8].

css: Incluye estilos provisionales para la aplicación, que luego habrán de ser modificados por el programado o diseñador para adecualos a los requisitos visuales impuestos por el cliente.

error: Contiene las páginas que se mostrarán en caso de error.

imagenes: Contiene las imágenes que se utilizan en la interfaz en botones, pulsadores de ordenación de tablas, etc.

js: Recopilación de útiles JavaScript que serán utilizados si los requisitos de accesibilidad de la aplicación lo permiten.

privado y publico: Ficheros JSP que implementan cada página de la interfaz de la aplicación, tanto aquellas zonas de acceso público como las de acceso restringido.

5.2.5. Capa de lógica de negocio - Fachada de Componentes (Modelo)

El modelo, ya sea implementado a partir de EJBs o POJOs, se encuentra en la siguiente ruta:

Estos archivos sólo representan la Fachada que se encarga de comunicar la lógica de negocio de la aplicación con los servlets, es decir, el modelo con el controlador.

Cuando se genera el esqueleto, Genereitor todavía no conoce qué entidades de la base de datos se van a manejar en la nueva aplicación, por lo que no es posible ofrecer estos archivos completos.

En su lugar, se ofrecen únicamente las cabeceras de los ficheros, y en la fase de *generación de código* se ofrecerán *trozos* que se insertarán en los ficheros que correspondan[9].

[8] web.xml es el descriptor de despliegue que indica al contenedor de servlets diversos parámetros, tales como la localización de los paquetes que forman el modelo, los descriptores de tags, las páginas que mostrar en caso de error, etc.

[9] Para automatizar la tarea de incrustar los trozos en los ficheros correspondientes se ha

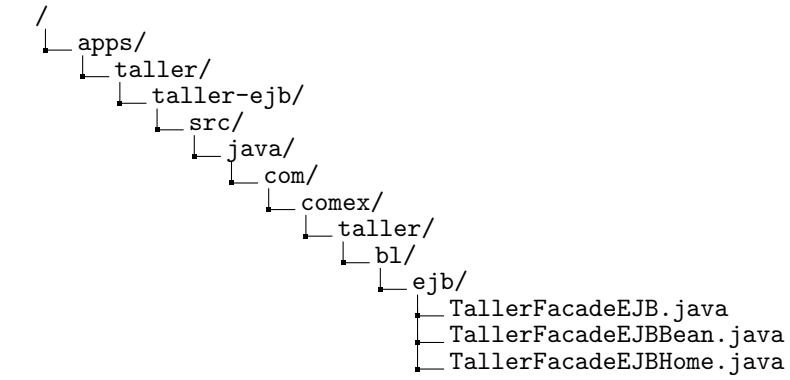

Figura 5.6: Ruta del modelo de una aplicación con EJB.

5.2.6. Capa de acceso a datos - Scripts SQL

Con la generación del esqueleto también se devuelven una serie de scripts SQL, aunque muchos de ellos estarán vacíos (recordemos que la implementación de la base de datos es un proceso **anterior** al uso de Genereitor). La ruta donde se encuentran es la de la figura 5.7:

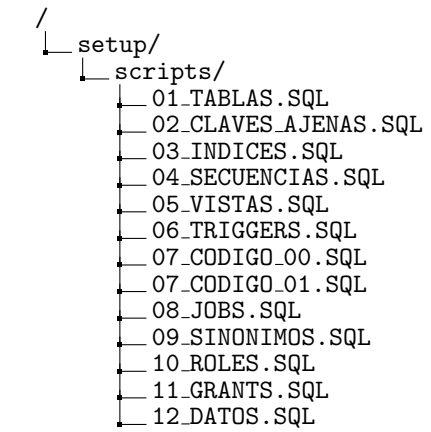

Figura 5.7: Ruta de los scripts SQL.

Tras la generación del esqueleto, estos ficheros únicamente contendrán los datos de creación de los componentes de la base de datos para el manejo de la tabla *APLICACION*_PARAMETRO, el rol *APLICACION*_ROL_APLICACION[10] y los *grants* (permisos) de este rol para modificar la tabla parámetro. Esta tabla será utilizada por la aplicación para guardar diversos parámetros.

Los ficheros 07_CODIGO_00.SQL y 07_CODIGO_01.SQL serán utilizados si la aplicación implementa su capa de acceso a datos mediante PL/SQL directamente en el servidor de bases de datos. El fichero 07_CODIGO_00.SQL contendrá los

implementado una herramienta auxiliar, Combineitor, cuyas características y funcionamiento se detallan en B en la página 145.

[10]Por ejemplo, TALLER_ROL_APLICACION.

tipos que utilizará PL/SQL mientras que 07_CODIGO_01.SQL contendrá la implementación de las funciones y procedimientos.

El estado inicial de 07_CODIGO_01.SQL será el siguiente:

```
1  CREATE OR REPLACE PACKAGE TALLER_PKG AS
2      TYPE tipo_cursor IS REF CURSOR;
3  --genereitor:insertarTrozo1
4  END TALLER_PKG;
5  /
6  CREATE OR REPLACE PACKAGE BODY TALLER_PKG AS
7  --genereitor:insertarTrozo2
8  END TALLER_PKG;
9  /
```

En este fichero se aprecian las trazas que luego serán utilizadas por combineitor para introducir los trozos de código generados mediante el bloque funcional de *generación de código*.

5.2.7. Documento de instalación

Por último, en la ruta setup/instalacion.html se proporciona un manual de instalación de la aplicación, a modo de guía para la instalación de la aplicación en el cliente.

Capítulo 6

Generación de partes de código

Este bloque funcional es el encargado de generar los componentes de la aplicación correspondientes a cada una de las entidades de la base de datos que van a formar parte de la aplicación que se va a desarrollar.

Generalmente será necesario realizar este paso para cada una de las entidades que forman parte de la base de datos, salvo aquellos casos que se decidan tener en cuenta las relaciones entre tablas, que como se verá más adelante, compartirán ciertos componentes de la aplicación.

También en este apartado se generarán *trozos* de los ficheros que quedaron incompletos en el apartado de generación de esqueleto, y que se introducirán en dichos ficheros mediante la herramienta Combineitor, cuyo funcionamiento está detallado en el apéndice B en la página 145.

6.1. Funciones de la interfaz

6.1.1. Fase 1: Conexión

El primer paso para la elaboración de las partes de código correspondientes a una entidad de la base de datos es la conexión al servidor de bases de datos. Por ello, la primera fase de este bloque funcional es la introducción de los datos de conexión:

- Sistema de bases de datos (lista desplegable con todos los sistemas de bases de datos soportados por la herramienta).
- Host.
- Puerto.
- SID, o nombre de la base de datos.

- Usuario.
- Contraseña.

También se proporciona un pulsador con el literal «*Default*», cuya función es autocompletar el siguiente bloque con los datos de conexión de las bases de datos de desarrollo que hay en la factoría de software de la empresa, según el servidor de bases de datos seleccionado. Con varias pulsaciones sobre este literal se rota de forma cíclica entre las diferentes conexiones para un mismo sistema de bases de datos, puesto que se da el caso que para algunos sistemas hay varias bases de datos de desarrollo.

Al pulsar el botón «Conectar», se realizan una serie de validaciones mediante JavaScript para comprobar que los datos introducidos tienen el formato correcto:

- *Sistema de bases de datos*, *Host*, *SID* y *usuario* no pueden tomar valores nulos.
- *Puerto* ha de tener valor numérico y no nulo.
- Si no se introduce un valor para el campo *contraseña*, se avisa al usuario mediante un mensaje junto al botón «Conectar», y se hace parpadear brevemente dicho campo para llamar su atención. Si se vuelve a pulsar el botón «Conectar» se lanzará efectivamente la conexión contra el servidor, indiferentemente de que se haya introducido una contraseña.

Si existe algún error en los datos se informará al usuario mediante una caja de texto situada al principio de la página. De lo contrario, se lanzará la conexión al servidor.

Es posible que los datos introducidos, pese a tener un formato válido, no sean correctos y el servidor de bases de datos no responda o rechace la conexión. De ser así, se informará nuevamente al usuario del error y se le permitirá modificar los datos introducidos para volver a lanzar la conexión[1].

6.1.2. Fase 2: Selección de tabla

En esta segunda fase se muestra al usuario, una vez se ha establecido con éxito la conexión a la base de datos, una lista con todas las tablas y vistas existentes en ella, de donde se elegirá la tabla correspondiente a la entidad para que se desea generar los componentes de la aplicación.

[1] Debido a que algunos sistemas de bases de datos ofrecen más información que otros cuando ocurre un error en la conexión, dependiendo del sistema elegido los textos explicativos del error serán más o menos específicos. Por ejemplo, mientras que un sistema puede devolver un error de tipo *usuario no válido*, otro lanzará simplemente *error en la conexión*.

6.1.3. Fase 3: Selección de tablas relacionadas

En caso de que en la base de datos existan tablas con claves ajenas que apunten hacia la tabla seleccionada en el paso anterior (ver figura 2.1 en la página 21, se muestran aquí dichas tablas relacionadas, ofreciendo así la posibilidad de generar la lógica para dichas tablas de manera conjunta, además de una presentación especial en las vistas que se generarán. Todo esto será explicado con detalle en la sección 6.2 en la página 58.

Esta vista se compone de dos listas, una que contendrá las tablas *seleccionables*, es decir, el conjunto de tablas cuyas relaciones apuntan a la tabla seleccionada en el paso anterior, y otra en la que se añadirán las tablas que el usuario quiere incluir en el paquete de código que va a ser generado.

Si no existen en la base de datos tablas cuyas claves ajenas referencien a la tabla elegida en el paso anterior, esta fase se omite, pasándose directamente a la fase 4.

6.1.4. Fase 4: Selección de campos

Ahora se presenta al usuario el listado de campos de la tabla (o tablas, en caso de haber seleccionado las relacionadas) elegida, en un formulario que ofrece varias opciones:

- Permite seleccionar qué campos aparecerán en las pantallas de listado de la aplicación web. Es posible que algunas propiedades de la entidad para la que se está generando código no interese que sean mostradas en las vistas de listado. Por esto, se permite al usuario seleccionar los campos que interesa que sean mostrados.
- Se pueden definir restricciones de obligatoriedad para los campos que no la tuvieran originalmente en la base de datos, ya por un error de diseño de ésta o cualquier otra circunstancia. Para los campos que sean clave primaria de la tabla, o que cuenten originalmente con dicha restricción impuesta directamente en la base de datos no será posible modificarla.
- También se podrá redefinir la longitud máxima que acepten los formularios de edición de la aplicación a generar. Si el programador introduce una restricción más permisiva que la establecida en la base de datos, se confiará en que acto seguido será modificada dicha restricción en la base de datos, por lo que Genereitor no reportará ningún tipo de error. Esto se ha decidido implementar de este modo, permitiendo *violar* las restricciones de tamaño porque no es raro encontrar errores de diseño en la base de datos, que habrán de ser subsanados aparte. De este modo, aunque la base de datos tuviera errores de este tipo en su diseño, se permite la generación del código acorde al diseño correcto, confiando en que el programador modificará la base de datos para subsanar el error.

  Las longitudes máximas aparecerán con los valores por defecto de las restricciones que establece la base de datos.

- Por último se permite definir la etiqueta (texto explicativo) que describirá cada propiedad de las entidades de la base de datos en la interfaz de la aplicación, por ejemplo en las descripciones de los campos de edición o buscador.

  Estos literales tendrán el valor por defecto del nombre de la propiedad de cada tabla, en *formato Java*[2].

También se ofrece al usuario información tal como el nombre de cada propiedad a la que hace referencia cada fila del listado y si es clave primaria de la tabla, así como el tipo de dato SQL que almacena y el tipo Java con el que será implementada la propiedad en la lógica de la aplicación.

6.1.5. Fase 5: Selección de parámetros

Esta última fase del bloque funcional de generación de código consta de tres partes:

6.1.5.1. Datos de aplicación

Como en los otros bloques funcionales, es necesario informar a Genereitor del nombre de la aplicación y el paquete base de la aplicación que se va a generar. Como en otras ocasiones en las que se requieren los datos, es requisito que el último componente del nombre de paquete sea el nombre de aplicación, por lo que no es necesario poner este último componente, siendo concatenado al final el nombre de la aplicación por la propia interfaz de genereitor.

6.1.5.2. Nombre de beans

Aquí se presenta la posibilidad de editar el nombre (o los nombres, en caso de haber seleccionado tablas relacionadas) de los beans que representan a las entidades de la base de datos.

Esta opción es debida a que con frecuencia las aplicaciones se desarrollan sobre bases de datos que no están implementadas por el equipo de desarrolladores de la empresa, sino que el propio cliente la proporciona (por ejemplo, en caso de una migración de una aplicación existente a otra nueva, conservando el modelo de datos de la anterior). Estas bases de datos es posible que no cumplan con las convenciones de nomenclatura que interesan al equipo de desarrolladores, por lo que aquí se ofrece la posibilidad de modificar el nombre que llevarán los objetos que representarán a las entidades de la base de datos.

Por ejemplo, es posible que en la base de datos que proporciona el cliente haya una tabla que contenga un listado de vehículos con sus características, llamada LISTADO_VEHICULOS. Genereitor propondrá como nombre del bean que represente a esta entidad ListadoVehiculos. Pero esto, aunque a nivel de implementación sería posible, no es lógico.

[2]La columna PROPIEDAD_DE_EJEMPLO tendría como literal por defecto PropiedadDeEjemplo.

Una tabla que represente un listado de vehículos en realidad almacenará entidades de tipo *vehículo* y no *listados de vehículos*, por lo que a la hora de implementar la aplicación es más lógico y más cómodo para el programador hablar de objetos de tipo Vehiculo.

Así, es posible modificar el nombre de los beans con los que se trabajará en la aplicación. No obstante, en los componentes de acceso a datos se seguirá refiriendo a las tablas por sus nombres originales (LISTADO_VEHICULOS, puesto que de lo contrario no sería posible acceder a la información.

6.1.5.3. Parámetros

En este último bloque de la página se permite al usuario seleccionar las características particulares de los componentes generados para la entidad sobre la que se está trabajando.

Módulos web: tal y como se vio en el bloque funcional de *generación de esqueleto*, una aplicación puede contar con varios módulos web que aprovechen la misma lógica de negocio, aunque tengan diferentes funcionalidades. Así, es probable que una entidad de la base de datos tenga que ser accesible por varios módulos web, por lo que aquí se ofrece una lista para añadir los módulos que han de implementar dicha entidad.

Aunque la entidad siempre estará disponible a nivel de modelo, porque todos los módulos web de la aplicación comparten esta capa, aquí se seleccionará para qué módulos se ha de implementar vista y controlador.

Utilizar Authenticator. Permite añadir un mecanismo de seguridad a las conexiones de la aplicación, consistente en un parámetro authenticator, que confirma que el usuario que solicita la conexión es un usuario válido conforme al registro de usuarios existente en la base de datos.

JSP para borrar varios beans en listado: genera, en las vistas de listado, la posibilidad de seleccionar varios elementos del mismo tipo –representados por el mismo tipo de bean– para ser borrados de forma simultánea, en lugar de tener que hacerlo uno por uno.

JSP de edición: Genera la parte de la vista correspondiente a la pantalla de edición de los objetos pertenecientes a la entidad de la cual se está generando la lógica, que permitirá añadir, editar o borrar registros en la base de datos.

JSP de detalle: Genera la parte de la vista correspondiente a la pantalla de detalle, que mostrará todas las propiedades de los objetos pertenecientes a la entidad.

Tratar las entidades secundarias en la misma página que la principal: Esta opción integra, en caso de que se hayan seleccionado tablas relacionadas a la principal, el listado de las secundarias en la misma pantalla que la edición de la entidad principal.

Por ejemplo, imaginemos que estamos trabajando con la entidad Propietario. Tendremos una entidad Vehiculo relacionada, puesto que un propietario puede tener varios, uno o ningún vehículos. Asumiremos que un vehículo siempre tiene propietario.

Esta opción nos permite añadir, a la vista de edición de la entidad Propietario el listado de entidades Vehiculo que le pertenecen, de forma que dicha lista de vehículos se refleja visualmente como una propiedad más del propietario.

Junto con el listado de vehículos se proporciona un buscador, de forma idéntica al listado de la entidad principal, Propietario.

Si no se selecciona esta opción, el listado de la entidad secundaria se mostrará en una pantalla diferente a la edición de la principal.

En la figura 6.1 se muestra el ejemplo de las entidades Propietario (principal) y Vehículo (relacionada) esquematizando las diferentes vistas que se generan para la gestión de los datos de ambas usando esta opción, es decir, con el listado de la secundaria en la edición de la principal. Por contra, en la figura 6.2 en la página siguiente se muestra el flujo de las vistas con esta opción desactivada.

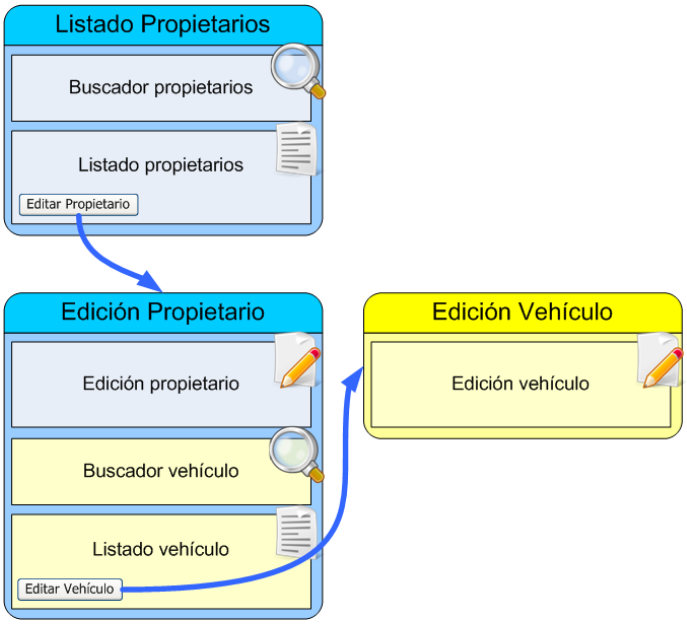

Figura 6.1: Esquema de las vistas con entidades relacionadas (amarillo) en la edición de la principal (azul).

EJB/Clases Java: Aquí seleccionará el programador si desea utilizar EJB o POJOs para implementar la capa de lógica de negocio, dependiendo de los rquisitos de la aplicación que se está desarrollando.

EJBs locales/remotos: Si se seleccionan EJBs para implementar la capa de lógica de negocio, aquí se seleccionará si se desea utilizar EJBs

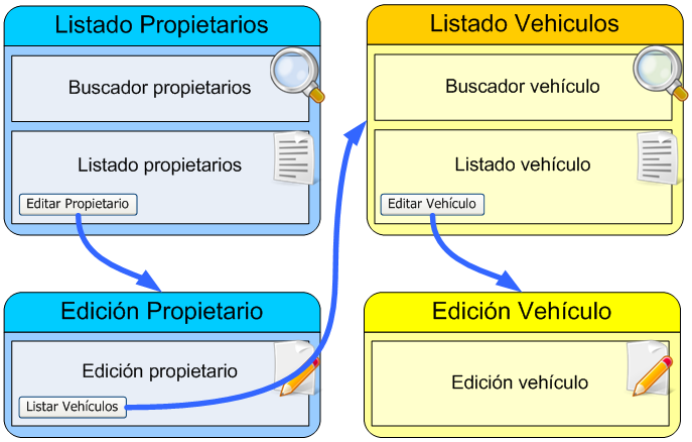

Figura 6.2: Esquema de las vistas con entidades relacionadas (amarillo) en diferente pantalla que la principal (azul).

remotos o locales[3].

PL/SQL o JavaSQL: Esta opción sólo está presente si se ha seleccionado una base de datos Oracle. PL/SQL permite implementar la lógica de acceso a datos en el propio servidor de bases de datos, liberando carga de trabajo al servidor de aplicaciones. De esta manera, los componentes de la capa de acceso a datos implementados en Java únicamente componen los filtros para las operaciones de acceso a datos y realizan las llamadas a las funciones implementadas en PL/SQL, que se ejecutan en el servidor de bases de datos y devuelven únicamente los datos requeridos.

Si se utiliza JavaSQL, las consultas a la base de datos se componen en la capa de acceso a datos, enviando peticiones a la base de datos y recibiendo y filtrando los resultados hasta obtener los registros deseados. Para accesos sencillos a la base de datos es indiferente el uso de un sistema u otro, pero cuando las consultas adquieren cierto grado de complejidad, es más eficiente utilizar PL/SQL si está disponible, puesto que los datos enviados entre el servidor de aplicaciones y el servidor de bases de datos son únicamente los que interesan, liberando a la red de tráfico innecesario.

6.2. Resultado

6.2.1. Capa web - JSP (Vista)

En la parte correspondiente a la vista, el paquete de código generado contendrá, según las opciones seleccionadas, los siguientes componentes:

[3]La diferencia entre EJBs remotos o locales es que los remotos son accesibles por todos los componentes de la aplicación, mientras que los locales únicamente pueden ser llamados desde la capa de lógica de negocio. Es por esto que normalmente la **fachada** se implementa como remota, ya que ha de ser accesible desde la capa web, y los demás componentes se implementan como locales, ya que sólo han de ser accedidos por la fachada y entre ellos.

6.2.1.1. JSP de listado

Será el fichero que se utilice para la composición dinámica de la vista de listado y buscador de la entidad generada.

Dicha vista se compondrá, en primer lugar, de un buscador que permite introducir valores para cualquiera de las propiedades de la entidad, y que al pulsar el botón «Buscar» devolverá una lista de los resultados filtrados por los valores introducidos.

Si existen en la entidad propiedades de tipo fecha, junto al campo para introducir dicho dato se proporciona un botón que al pulsarlo despliega un calendario, permitiendo al usuario de la aplicación seleccionar el día con mayor comodidad. Este calendario está implementado con JavaScript y CSS, por lo que no funcionará en un navegador sin soporte para JavaScript. No obstante, se podrá introducir la fecha directamente en la casilla correspondiente, por lo que la accesibilidad se mantiene.

Si en la entidad existen propiedades de tipo *archivo binario* (BLOB), no se ofrece la posibilidad de filtrar por dichas propiedades, ya que sería un absurdo.

Si la clave primaria de la entidad es **única** (consta de un sólo campo) y **numérica** se asume que es autonumérica y por tanto no se ofrece buscar por dicho campo, puesto que no tendría utilidad filtrar por una propiedad de tales características. La suposición de que el campo es autonumérico es necesaria, puesto que en algunos sistemas de bases de datos no existe el tipo de datos SERIAL (e.g. Oracle), por lo que no es posible determinar consultando los metadatos de la base de datos si dicho campo es autonumérico o realmente es un código numérico con algún significado. Es por esto por lo que se proporciona también la alternativa para considerar dicha clave primaria como campo numérico, en forma de comentarios en la página.

Bajo el buscador se encuentra el listado. Dicho listado se construye con un tag tabla proporcionado con la generación del esqueleto, que generará dinámicamente el listado a partir de un objeto BaseBeanSet[4] que recibe como parámetro.

Dicho tag tabla recibe una serie de descriptores, tanto de tabla como de columna, que indican de qué manera ha de dibujarse (por ejemplo, el número y características de las columnas que formarán el listado).

Para cada entidad de la base de datos del tipo listado que cumpla con las condiciones impuestas por el filtro de búsqueda –en caso de haberse especificado– se mostrará una fila en la tabla. Si el número de filas a mostrar excede de 10^5, el paginador de la tabla se encargará de mostrar sólo los diez primeros elementos, creando al pie de la tabla una lista de *páginas* de la tabla, a las que se puede acceder pinchando directamente sobre los números o sobre las flechas de *anterior* y *siguiente*. Este paginador también permite ordenar la tabla por cualquiera de las propiedades de la entidad que aparecen, tanto ascendente como descendentemente. Esto se realiza pinchando sobre las flechas que hay en la

[4] BaseBeanSet es una lista de objetos que representan una entidad en la base de datos. En el caso del listado, contendrá todos los objetos de la base de datos del tipo que se quiere listar.

[5] 10 es el valor por defecto que se ofrece para la longitud de las páginas, pero es totalmente personalizable.

cabecera de cada columna. Al realizar una ordenación, se resaltará la columna por la que se ha ordenado la tabla, y el sentido, mediante el cambio de color de la cabecera de dicha columna y las flechas.

Para cada registro de la base de datos se mostrará además:

- Un *checkbox* para seleccionar registros y poder borrarlos posteriormente con el botón inferior de «borrar», en caso de que se haya seleccionado a la hora de generar el código la opción «generar código para borrar varios beans en listado».
- Un botón de edición, que llevará a la página de edición de la entrada cuyo botón se pulse, permitiendo al usuario modificar el registro, en caso de que se haya seleccionado al generar el código la opción «generar JSP de edición».
- Si existen propiedades correspondientes a archivos binarios (BLOBs) se muestran, por cada uno, dos botones:

  Descargar binario: Permite descargar el binario almacenado en la base de datos al ordenador del usuario. Si hay algún fichero almacenado en el registro, el botón aparecerá de color naranja (*activo*) y al pulsarlo comenzará la descarga[6]. Si dicho registro no almacena un binario, el botón se muestra gris (*inactivo*) y se avisa al usuario de que no hay datos almacenados al pulsarlo.

  Borrar binario: Permite borrar el fichero binario del registro seleccionado.

Bajo el listado se ofrecen también los botones de «Borrar», que elimina los registros seleccionados en la tabla, «Nuevo», que lleva a la vista de edición de la entidad mostrando los registros en blanco para la adición de un nuevo registro, y «Volver», que permite regresar a la página visitada anteriormente.

Todos los estilos utilizados, tanto en el buscador como en el listado, están definidos en la hoja de estilos CSS proporcionada con la generación de esqueleto.

6.2.1.2. JSP de edición

Al editar o insertar un registro, el controlador llama a la vista de edición.

En esta pantalla se presentan al usuario casillas para rellenar con los datos oportunos el nuevo registro, o modificar los ya existentes en caso de tratarse de una edición.

Nuevamente si existe un campo de tipo fecha, se proporciona el calendario junto a la casilla correspondiente.

[6]Como se explicará más adelante, tanto las cargas como las descargas de binarios se hacen mediante *streaming*, lo que evita saturar la memoria del servidor de aplicaciones, a la vez que permite el uso de, por ejemplo, reproductores multimedia, de forma que no tengan que esperar a la descarga del archivo completo para comenzar la reproducción del recurso, sino que en cuanto acabe la descarga del primer «trozo» de fichero comenzará, almacenando en el búfer los datos de los siguientes trozos conforme se vayan descargando del servidor.

Si la clave primaria es **única** y **numérica** se asume que es autonumérica, por lo que no se muestra el campo para añadir dicho dato. Se asume que al realizar la inserción le será asignado un nuevo valor de clave primaria al registro.

Si existen campos binarios (BLOBs) el formulario proporcionado es de tipo *multipart*, que permite subir ficheros grandes en partes al servidor. Si es una operación de edición, tanto si hay binarios almacenados como si no, aparecen los campos de selección de archivo vacíos. Si se modifican, se añadirá o sobreescribirá el binario de la base de datos por el subido recientemente. Si se dejan en blanco, no se modificarán dichos campos en la base de datos. Para la operación de eliminar el binario almacenado en un registro, se proporciona el botón «Borrar binario» en la pantalla de listado.

Al final de la página se proporcionan los botones de «Aceptar» y «Volver», que retornarán al usuario a la pantalla de listado, guardando o descartando respectivamente las modificaciones.

Entidades relacionadas

En caso de que se haya generado componentes para las tablas relacionadas, en esta página aparecerán además:

Buscador y listado de cada entidad relacionada si se seleccionó la opción de tratar entidades relacionadas en la misma página que la principal (figura 6.1 en la página 57).

Botones de listado de entidades secundarias si no se seleccionó dicha opción, que llevará a la vista de listado de la entidad secundaria (figura 6.2 en la página 58).

En ambos casos el buscador y listado tendrán idéntico comportamiento que los de la entidad principal, pero mostrarán únicamente los campos correspondientes a dicha entidad principal. Por ejemplo, en el caso de *vehículos y usuarios*, al editar un usuario y mostrar la lista de vehículos, sólo se listarán los vehículos correspondientes a dicho usuario, y no los de los demás.

Validaciones de datos introducidos

Las validaciones de los datos introducidos se realizan por defecto en la máquina del cliente, mediante el intérprete de JavaScript del usuario. Si los requisitos de accesibilidad de la aplicación impiden el uso de dicho sistema, se han de eliminar estas validaciones y descomentar las ofrecidas en los servlets.

Etiquetas

Todos los textos que figuran en la interfaz son generados mediante etiquetas, mediante el tag etiqueta proporcionado con el esqueleto, generadas en un *trozo* del fichero etiquetas.properties que se entrega con cada paquete de código.

6.2.2. Capa web - Servlet (Controlador)

El componente correspondiente al controlador que manejará las peticiones del usuario, provinientes de la vista, e interactuará con el modelo, consultando o modificando los datos, se proporciona también en el paquete de código generado para cada entidad.

El servlet proporcionado contará con las siguientes funciones[7]:

perform(), que es la encargada de recibir la llamada de la vista y dirigir la petición del usuario hacia la acción del servlet que corresponda.

inicio(), es la función inicial del controlador de la entidad. Se encarga de seleccionar la vista de listado, tras establecer un filtro de búsqueda nulo (para que la lista muestre todos los registros) y una ordenación por defecto. Para obtener la lista de elementos llamará a la función getLista*Entidad*() de la fachada del modelo.

listar(), se encarga de crear la vista de listado tras haber insertado, actualizado o borrado un registro. Mantiene la ordenación, la paginación y el filtro de búsqueda establecidos antes de comenzar el proceso de edición, inserción o borrado. Para obtener la lista de elementos llamará a la función getLista*Entidad*() de la fachada del modelo.

listarSession(), de cometido similar a listar(), salvo que devuelve la vista de listado con la paginación, ordenación y filtro de búsqueda definidos por defecto en la sesión. Para obtener la lista de elementos llamará a la función getLista*Entidad*() de la fachada del modelo.

paginar(), establece el cambio de orden o la agrupación en páginas de n elementos (10 por defecto). Define la propiedad de la entidad por la que se ordena, el tipo de ordenación (ascendente o descendente), y el número de página que va a mostrar la vista. Tras componer la lista de elementos ordenada, se invoca a la vista de listado. Para obtener la lista de elementos llamará a la función getLista*Entidad*() de la fachada del modelo.

buscar(), crea un filtro de búsqueda a partir de los parámetros introducidos en el formulario de buscador, y devuelve al usuario la lista de elementos que coincidan con dicho filtro. Para obtener la lista de elementos llamará a la función getLista*Entidad*() de la fachada del modelo.

nuevo(), Esta función únicamente invoca a la vista de edición de la entidad correspondiente. En caso de que la tabla de la base de datos que representa dicha entidad tenga clave primaria **única** y **numérica** no aparece el campo correspondiente a esta propiedad, ya que se asume que es autonumérica y la capa de acceso a datos le asignará automáticamente un valor.

[7]Nota: en los nombres de las funciones, el literal *Entidad* equivale al nombre de la entidad que se introdujo en la etapa de selección de parámetros del bloque funcional de generación de código. Por ejemplo, la función getEntidad() equivaldrá a getPropietario() en el ejemplo de las entidades Propietario y Vehículo ya usado.

Asímismo, el literal Propiedadbinaria equivale al nombre de la propiedad cuyo tipo de datos almacenados es BLOB. Por ejemplo, getFotografiaCatalogo().

insertar(), Esta función, que será llamada desde la vista de edición una vez se hayan rellenado los campos correspondientes a las propiedades de la entidad, llamará a la función insert*Entidad*() de la fachada del modelo, pasando un objeto del tipo de la entidad.

editar(), invoca la vista de edición, con los valores de todas las propiedades del objeto que se quiere editar, para que el usuario modifique las que desee. Al igual que la función insertar(), si la clave primaria de la entidad es **única** y **numérica** no se recoge dicho valor, puesto que no será posible modificarlo.

actualizar(), al confirmar los cambios, esta función envía el objeto modificado a la función update*Entidad*() de la fachada de la capa de lógica de negocio.

borrar(), recoge el valor (o valores) de la clave primaria del registro y llama a la función delete*Entidad*() de la fachada.

borrarVarios(), recoge el valor de la clave primaria de los registros seleccionados y llama a la función deleteLista*Entidad*() de la fachada de la capa de lógica de negocio.

createDescriptorTabla(), que se encarga de crear los descriptores de la tabla y las columnas de los listados. Esta función es llamada por las anteriores cada vez que se ha de presentar un listado en pantalla.

createEntidad(), que se encarga de recoger los valores de la vista para crear un objeto del tipo correspondiente a la entidad con la que se trabaja.

getSelected(), averigua las claves primarias de los registros que están seleccionados en el listado.

tratarError(), define el comportamiento de la aplicación en caso de que se produzca algún error.

6.2.2.1. Servlets de entidades con binarios

En caso de que en la entidad para la que se ha generado código posea campos que almacenan binarios (de tipo BLOB), además de las funciones anteriores se generan las siguientes:

getPropiedadbinariaEntidad(), que llamará a la función get*PropiedadbinariaEntidad*() de la fachada y devolverá al usuario el archivo binario almacenado en la base de datos. La descarga del fichero se efectúa mediante *streaming*, es decir, se envían los trozos concatenados en vez de cargar el fichero de una vez, lo que mejora el rendimiento y evita el colapso del servidor de aplicaciones por desbordamiento de memoria.

delPropiedadbinariaEntidad(), que llamará a la función del*PropiedadbinariaEntidad*() de la fachada borrando el binario almacenado.

createEntidadMultipart(), cuya función es recuperar los atributos del registro que se introduce en las pantallas de inserción o actualización, ya que en caso de haber binarios los formularios de dichas pantallas son de tipo *multipart*. La función create*Entidad*() no tiene en cuenta los campos binarios.

También las funciones insertar() y actualizar() sufren cambios al aparecer atributos binarios, puesto que se subirán por partes al servidor de bases de datos: primero se almacenan los atributos no binarios, y luego se divide el binario en trozos y mediante la función app*PropiedadbinariaEntidad* de la fachada se envían los fragmentos al servidor.

Esta función optimiza la memoria consumida en el servidor de aplicaciones, evitando que se colapse por *overflow*.

6.2.2.2. Servlets de entidades relacionadas

En caso de que se hubieran seleccionado tablas relacionadas en la fase de selección de parámetros en el bloque funcional de generación de código de Genereitor, aparecerán por cada entidad secundaria funciones idénticas a las de la principal, pero incluirán el nombre de la entidad a la que ofrecen funcionalidad. Por ejemplo, en el caso de Vehiculo, que es entidad relacionada de Propietario:

insertar() corresponderá a la inserción de una entidad Propietario, es decir, la entidad *principal*.

insertarVehiculo() corresponderá a la inserción de una entidad Vehiculo, es decir, la entidad *relacionada*.

6.2.3. Capa de lógica de negocio - Fachada

Aunque los ficheros de fachada ya se proporcionan con la generación de esqueleto, se van completando conforme se suceden las generaciones de código de las diferentes entidades de la base de datos.

El paquete de código generado para una entidad contiene dos *trozos* de la fachada. En caso de que la aplicación utilice EJBs:

- Trozo de *Aplicacion*FacadeEJB.
- Trozo de *Aplicacion*FacadeEJBBean.

Y si se utilizan POJOs:

- Trozo de FacadeBL.
- Trozo de FacadeBLImpl.

En conjunto, la fachada contiene las llamadas a todas las funciones disponibles en la lógica de negocio de la aplicación.

6.2.4. Capa de lógica de negocio - Componentes

Los componentes de la capa de lógica de negocio, ya sean EJB o POJO, implementan la forma en la que los datos han de ser tratados, y realizan las llamadas a la capa de acceso a datos, que será la que interactúe con la base de datos para obtener o modificar la información contenida en ella.

También manejan la apertura y clausura de las instancias de la conexión con el servidor de bases de datos, y de ser necesario, implementan soporte transaccional para dichas conexiones.

La gestión transaccional garantiza que la base de datos va a mantener siempre un estado íntegro y consistente, de manera que si falla la ejecución de una operación, se devuelve la base de datos al estado inmediatamente anterior al comienzo de la ejecución de dicha operación (*rollback*), de modo que no queden operaciones *a medias* que pongan en peligro la integridad de la base de datos.

Las operaciones implementadas en dichos componentes serán, para cada entidad de la base de datos:

getEntidad(): Llama a la capa de acceso a datos para que devuelva un objeto del tipo de la entidad, con todas sus propiedades. Como parámetro se le ha de pasar un objeto del tipo de la entidad con, al menos, el valor de los campos que compongan la clave primaria. De otra forma, lanzará una excepción.

getListaEntidad(): Llama a la capa de acceso a datos para que devuelva una lista de todos los registros contenidos en ella. También se puede pasar un filtro del tipo de la entidad, para que devuelva sólo los registros que coincidan con los criterios que dicho filtro impone.

deleteEntidad(): Llama a la función deleteListaEntidad() tras crear una lista de un sólo elemento, que será el que tenga por clave primaria la del filtro recibido como parámetro.

deleteListaEntidad(): Elimina todos los registros de la base de datos que coincidan con los criterios establecidos por el filtro que se recibe como parámetro. Este filtro no tiene por qué poseer un valor de clave primaria.

insertEntidad(): Llama a la capa de acceso a datos para que inserte el registro que corresponde al filtro que se recibe como parámetro. Si la clave primaria de la entidad es **única** y **numérica**, retornará el valor de la clave primaria que la capa de acceso a datos ha asignado al registro, ya que se asumirá autonumérica.

updateEntidad(): Llama a la capa de acceso a datos para que actualice el registro con clave primaria la del filtro que recibe como parámetro, con los atributos de dicho filtro.

Gestión transaccional y de conexiones:

beginTransaction(): marca el comienzo de la transacción (sólo EJB).

commitTransaction(): finaliza y confirma el éxito de la transacción (sólo EJB).

rollbackTransaction(): devuelve la transacción al estado que tenía al hacer beginTransaction() (sólo EJB).

getConnection(): abre una conexión con la base de datos.

closeConnection(): finaliza la conexión.

En caso de que la entidad contenga campos binarios, se proporcionan además:

appendPropiedadbinariaEntidad(): Llama a la capa de acceso a datos para que inserte un trozo del binario en un registro. La clave primaria y el trozo del binario son atributos del filtro que recibe como parámetro.

getPropiedadbinariaEntidad(): Consulta a la capa de acceso a datos el contenido del binario almacenado en la propiedad correspondiente a esta función.

deletePropiedadbinariaEntidad(): Llama a la capa de acceso a datos para que elimine el binario almacenado en la propiedad correspondiente.

6.2.5. Capa de lógica de negocio - Excepciones

Para cada entidad sobre la que se genera código, se proporciona también la implementación de sus excepciones. Evidentemente, dicha implementación será muy básica, puesto que no se pueden conocer de forma automática las situaciones en las que la entidad podrá causar excepciones, ni qué desea hacer el programador para manejarlas.

Por ejemplo, la implementación de la excepción para una entidad Catálogo será la siguiente:

```
1  package com.comex.taller.bl.exception;
2
3  import com.comex.common.exception.BaseException;
4
5  public class CatalogoException extends BaseException {
6
7    /**
8     * Constructor por defecto. El error sera ERROR_INDEFINIDO
9     * @param texto El texto de la exception
10    */
11   public CatalogoException(String texto) {
12     super(texto);
13   }
14
15   /**
16    * Constructor con codigo de error
17    * @param texto El texto de la exception
18    * @param codigoError El codigo de error de la exception
19    */
20   public CatalogoException(String texto, int codigoError) {
21     super(texto, codigoError);
22   }
23
24   /**
25    * Constructor con codigo de error
26    * @param codigoError El codigo de error de la exception
27    */
28   public CatalogoException(int codigoError) {
29     super(codigoError);
30   }
31 }
```

Como se aprecia, hereda las funciones de la superclase BaseException a modo de excepción genérica, pero se contempla la posibilidad de personalizar su comportamiento.

6.2.6. Capa de acceso a datos - Componentes

Las clases Java correspondientes a la capa de acceso a datos serán las encargadas de comunicarse con la base de datos, y contendrán las consultas o modificaciones necesarias para realizar su tarea.

Las funciones implementadas en la capa de acceso son llamadas por los componentes de la lógica de negocio (EJBs o POJOs), y hacen uso de la API jdbc que proporciona J2EE para comunicarse con la base de datos.

Implementan las siguientes funciones:

getEntidad() a partir del filtro proporcionado, que habrá de tener asignados al menos los valores de la clave primaria, devuelve todas las propiedades del registro.

getListaEntidad() devuelve todos los registros de la entidad que coincidan con los criterios de búsqueda que establece el filtro.

insertEntidad() inserta un registro de la entidad, con los valores del filtro. Si la entidad tiene clave primaria **única** y **numérica** se asume que es autonumérica, no se pasará valor para la clave primaria, sino que se recibirá de la base de datos al insertar el registro, asumiendo que el sistema de base de datos le asignará dicho valor al registro. El valor de la clave primaria se devuelve al completar con éxito la operación. Si la clave primaria no es autonumérica, la función no devuelve nada.

insertListaEntidad() esta función no se usa originalmente, simplemente admite un BaseBeanSet de filtros y por cada elemento de dicho set, llama a la función insertEntidad().

updateEntidad() actualiza el registro de la entidad con los valores del filtro.

updateListaEntidad() esta función no se usa originalmente, simplemente admite un BaseBeanSet de filtros y por cada elemento de dicho set, llama a la función updateEntidad().

deleteEntidad() crea un BaseBeanSet de una posición, que contendrá el filtro que acepta como parámetro, y llama a deleteListaEntidad().

deleteListaEntidad() borra los registros cuyas claves primarias coincidan con las de los filtros contenidos en el BaseBeanSet que acepta como parámetro.

Las funciones insertListaEntidad() y updateListaEntidad() no se utilizan en la aplicación generada, puesto que los casos en los que se quiera insertar varios registros en una misma operación son escasos. De todos modos se ofrecen por si en un futuro se decidiera dotar a la aplicación de esta posibilidad.

Tampoco se utiliza deleteEntidad(), ya que el borrado se suele hacer por lista de varios elementos, borrando los que el usuario seleccionó en el listado de la vista. Esta función se mantiene por si el desarrollador decide añadir la funcionalidad de borrado unitario. No obstante, dicha operación se ejecuta actualmente con deleteListaEntidad(), donde la lista será de un único elemento.

Acceso a datos de entidades con binarios

Aparte de las funciones ya expuestas, para aquellas entidades que tengan alguna propiedad de tipo binario (BLOB), se añadirán las siguientes funciones:

appendPropiedadbinariaEntidad() inserta en la base de datos el objeto binario en la propiedad de la entidad correspondiente, de manera que si el tamaño total del binario excede de la longitud establecida (512kb por defecto), añadirá sólo un trozo del fichero. Esta función será llamada recurrentemente hasta completar la totalidad del binario.

getPropiedadbinariaEntidad() obtiene el objeto binario de la base de datos y lo devuelve a la fachada.

deletePropiedadbinariaEntidad() elimina el objeto binario de la base de datos.

Estas funciones corresponden a una única propiedad de la entidad, de manera que si ésta contase con varias propiedades binarias, se generarían las funciones para cada una de dichas propiedades.

Acceso a datos con entidades relacionadas

En caso de que se hayan seleccionado entidades relacionadas a la tabla principal, se generará una implementación diferente para cada entidad, tanto principal como relacionadas, al contrario que en los servlets, en los que se generaban acciones para todos los objetos dentro del mismo componente.

6.2.7. Capa de acceso a datos - PL/SQL

Si el sistema de bases de datos elegido para la nueva aplicación es Oracle y se ha seleccionado PL/SQL como método de acceso a datos, se generarán las funciones correspondientes en dicho lenguaje para la entidad –o entidades– seleccionada.

Un problema del servidor de bases de datos Oracle es que no acepta nombres de funciones, procedimientos o identificadores de más de 30 caracteres. Por esto, se realiza un *recorte* de los nombres de las tablas en la base de datos, a partir de los cuales se crean dichos identificadores, de forma que queden reconocibles por el usuario y no excedan de la longitud máxima permitida.

Este proceso de recorte de los identificadores de función se representa en la figura 6.3 en la página siguiente.

Figura 6.3: Proceso de recorte de identificadores en funciones PL/SQL

Por ejemplo, imaginemos que tenemos una tabla APLICACION_TABLA_DE_NOMBRE_EXTREMADAMENTE_LARGO. El nombre de la función PL/SQL original sería APLICACION_FN_INSERT_ TABLA_DE_NOMBRE_EXTREMADAMENTE_LARGO, donde:

APLICACION será el nombre de la aplicación para la que se está generando el código PL/SQL.

FN indica que el código que sigue corresponde a una función PL/SQL. En dicho lenguaje hay funciones y procedimientos, que se diferencian en que las primeras retornan un valor, mientras que los últimos no. Un procedimiento se identificará con el literal PR.

INSERT es el identificador de la tarea que realiza esta función, en este caso

insertar.

TABLA_DE_NOMBRE_EXTREMADAMENTE_LARGO es el nombre de la tabla.

Vemos que el identificador de la función generado excede de la longitud máxima permitida, por lo que habrá que recortarla.

APLIC$_{5}$ACION$_{10}$_FN_I$_{15}$NSERT$_{20}$_TABL$_{25}$A_DE_$_{30}$NOMBR$_{35}$E_EXT$_{40}$ REMAD$_{45}$AMENT$_{50}$E_LAR$_{55}$GO, 57 caracteres en total.

El primer paso será suprimir los guiones bajos del nombre de la tabla:

APLIC$_{5}$ACION$_{10}$_FN_I$_{15}$NSERT$_{20}$_TABL$_{25}$ADENO$_{30}$MBREE$_{35}$XTREM$_{40}$ ADAME$_{45}$NTELA$_{50}$RGO, 53 caracteres en total.

Como el nombre sigue siendo demasiado largo, recortaremos los componentes del nombre de la tabla (*TABLA*, *DE*, *NOMBRE*, *EXTREMADAMENTE* y *LARGO*) empezando por el primero y dejándolos con una longitud mínima de tres caracteres:

APLIC$_{5}$ACION$_{10}$_FN_I$_{15}$NSERT$_{20}$_TABD$_{25}$ENOMB$_{30}$REEXT$_{35}$REMAD$_{40}$ AMENT$_{45}$ELARG$_{50}$O, tras recortar «TABLA» convirtiéndolo en «TAB». «DE» tiene menos de 3 caracteres, así que no se recorta.

APLIC$_{5}$ACION$_{10}$_FN_I$_{15}$NSERT$_{20}$_TABD$_{25}$ENOMB$_{30}$REREE$_{35}$XTREM$_{40}$ ADAME$_{45}$NTELA$_{50}$RGO, tras recortar «NOMBRE» convirtiéndolo en «NOM». Tras recortar todos los componentes:

APLIC$_{5}$ACION$_{10}$_FN_I$_{15}$NSERT$_{20}$_TABD$_{25}$ENOME$_{30}$XTLAR$_{35}$, 35 caracteres en total.

Tras finalizar los recortes todavía se excede la longitud permitida, por lo que, como último recurso, se trunca el nombre de la función hasta el máximo número de caracteres permitidos:

APLIC$_{5}$ACION$_{10}$_FN_I$_{15}$NSERT$_{20}$_TABD$_{25}$ENOME$_{30}$, 30 caracteres en total, por lo que ya es una longitud válida.

A partir de estos identificadores *recortados* se creará el código necesario para generar el script PL/SQL que, una vez compilado en el servidor de bases de datos, se convertirá en las funciones y procedimientos disponibles para realizar el tratamiento de los datos.

Las funciones y procedimientos generados son:

GET_ENTIDAD, función que recibe la clave primaria de la entidad y devuelve todas sus propiedades.

GET_ENTIDADES, función que recibe un filtro del tipo de la entidad, y devuelve todos los registros que coinciden con los criterios de búsqueda establecidos en el filtro.

INSERT_ENTIDAD, añade un registro del tipo de la entidad a la base de datos. Dependiendo de si su clave primaria es autonumérica o no, será función –y devolverá el valor de la clave autonumérica del registro que se acaba de

añadir– o procedimiento –no devolverá nada, puesto que la clave primaria será proporcionada con el resto de parámetros de entrada–.

UPDATE_ENTIDAD, procedimiento que modifica las propiedades de un registro de la base de datos.

DELETE_ENTIDADES, procedimiento que elimina de la base de datos los registros cuya clave primaria coincida con las de la lista que se pasa por parámetro.

Si la entidad para la que se está generando código posee propiedades que almacenan objetos binarios (BLOB), para cada una de dichas propiedades se generarán también las siguientes funciones y procedimientos:

APPBLOBn_ENTIDAD, que será el procedimiento encargado de añadir los *trozos* del fichero binario al registro de la base de datos.

GETBLOBn_ENTIDAD, función que devolverá el objeto binario almacenado en el registro.

DELBLOBn_ENTIDAD, procedimiento que asigna valor NULL a la propiedad binaria del registro.

Como en una entidad puede haber varios campos binarios y es necesario disponer de estas tres funciones para cada uno de estos campos, el nombrado de dichas funciones se realiza de forma diferente a las anteriores.

Se otorgan los nombres de las funciones numerándolas (APPBLOB1_ENTIDAD para el primer campo binario, APPBLOB2_ENTIDAD para el segundo, etc), en lugar de incluir el nombre de la propiedad sobre la que actúan. Esto es debido a la restricción de 30 caracteres de Oracle. Por ello, antes de la implementación de estas funciones se señala en forma de comentario a qué parámetro representan.

6.2.8. Capa de acceso a datos - Beans

Un bean es un objeto Java que representa a una entidad de la base de datos. Una entidad tiene una serie de propiedades, que estarán en consecuencia representadas en el bean.

Aparte de las propiedades que tiene la propia entidad en la base de datos, es común que en la implementación del bean aparezcan otras propiedades adicionales, que enriquecen el objeto.

Por ejemplo, en el caso de las entidades Propietario y Vehículo, visto en la figura 2.1 en la página 21, se observa que en ambos casos hay funciones (señaladas en negrita) que no se corresponden con ninguno de los parámetros de las tablas, sino que representan la relación entre ambas (listaVehículos en propietario, y propietario en vehículo) Estas propiedades son en sí mismas objetos dentro de los objetos, que serán utilizadas por la capa de lógica de negocio cuando se necesite analizar dichas relaciones entre tablas, pero no se utilizarán a la hora de leer o escribir en la base de datos.

Otro ejemplo es el caso de las tablas con BLOBs, como la entidad Catálogo:

TALLER_CATALOGO	
ID_CATALOGO	**NUMBER**
ID_VEHICULO	NUMBER
FOTOGRAFIA	BLOB
DESCRIPCION	CLOB

Sin embargo, se observan los siguientes parámetros en la implementación del bean:

```
1       public static final String TABLA = "TALLER_CATALOGO";
2       public static final String SECUENCIA = "TALLER_SQ_CATALOGO";
3       protected Long idCatalogo;
4       protected Long idVehiculo;
5       protected byte [] fotografia;
6       protected String pathFotografia;
7       protected Integer lengthFotografia;
8       protected String descripcion;
```

Aparte de las dos constantes que aparecen al principio, que señalan la tabla que representa a esta entidad en la base de datos y la secuencia que le corresponde (es una tabla con clave primaria autonumérica), existen dos propiedades *extra*, pathFotografia y lengthFotografia, cuyos valores no se almacenan en la base de datos, sino que son utilizados por la lógica de la apliación para desempeñar diferentes funciones.

Las funciones que implementa cada bean son las siguientes:

Constructora vacía: inicializa un objeto del tipo la entidad correspondiente con todos sus valores nulos.

clone: crea una copia (clon) de un objeto del tipo la entidad correspondiente.

get de cada propiedad: devuelve el valor de dicha propiedad, en su tipo de datos primario si lo hay (e.g. int).

getObject de cada propiedad: devuelve el valor de dicha propiedad en su tipo de datos objeto (e.g. Integer).

set de cada propiedad: establece el valor de dicha propiedad, en su tipo de datos primario si lo hay (e.g. int).

setObject de cada propiedad: establece el valor de dicha propiedad en su tipo de datos objeto (e.g. Integer).

Ejemplo de algunas funciones correspondietes al bean Catalogo serán:

Listing 6.1: Catalogo.java

```
1   public Catalogo(){
2      idCatalogo = null;
3      idVehiculo = null;
4      fotografia = null;
5      descripcion = null;
6      pathFotografia = null;
7      lengthFotografia = null;
8   }
9
```

```
10  public Object clone(){
11     Catalogo nuevo = (Catalogo) super.clone();
12     nuevo.idCatalogo = idCatalogo;
13     nuevo.idVehiculo = idVehiculo;
14     nuevo.fotografia = fotografia;
15     nuevo.descripcion = descripcion;
16     nuevo.pathFotografia = pathFotografia;
17     nuevo.lengthFotografia = lengthFotografia;
18     return nuevo;
19  }
20
21  /**
22  * Devuelve el campo idVehiculo como Object
23  * @return idVehiculo como Object
24  */
25  public Long getIdVehiculoObject() {
26     return idVehiculo;
27  }
28
29  /**
30  * Informa el campo idVehiculo como Object
31  * @param idVehiculo El nuevo valor del campo idVehiculo
32  */
33  public void setIdVehiculoObject(Long idVehiculo) {
34     this.idVehiculo = idVehiculo;
35  }
36
37  /**
38  * Devuelve el campo idVehiculo como tipo basico
39  * @return idVehiculo como tipo basico
40  */
41  public long getIdVehiculo() {
42     if (idVehiculo != null)
43        return idVehiculo.longValue();
44     else
45        return (long)0;
46     }
47
48  /**
49  * Informa el campo idVehiculo como tipo basico
50  * @param idVehiculo El nuevo valor del campo idVehiculo
51  */
52  public void setIdVehiculo(long idVehiculo) {
53     this.idVehiculo = new Long(idVehiculo);
54  }
```

Capítulo 7

Generación de script para *tablas maestras*

Las tablas maestras son aquellas que únicamente contienen información respecto a la entidad a la que representan. Por tablas maestras se entienden tablas sencillas, que generalmente serán de tipo identificador→valor y, en caso de mantener alguna relación con otra tabla, una clave ajena.

Por ejemplo, en el caso de la figura 7.1, las tablas únicamente contienen un identificador y un nombre, y en caso de la tabla MUNICIPIO y PROVINCIA, la clave ajena que referencia a la PROVINCIA y COMUNIDAD_AUTONOMA a la que pertenecen respectivamente.

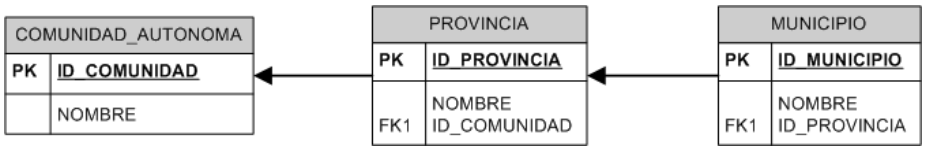

Figura 7.1: Ejemplo de tablas maestras relacionadas.

7.1. Funciones de la interfaz

7.1.1. Fase 1: Conexión

El primer paso para la elaboración de las partes de código correspondientes a una entidad de la base de datos es la conexión al servidor de bases de datos. Por ello, la primera fase de este bloque funcional es la introducción de los datos de conexión:

- Sistema de bases de datos (lista desplegable con todos los sistemas de bases de datos soportados por la herramienta).

- Host.
- Puerto.
- SID, o nombre de la base de datos.
- Usuario.
- Contraseña.

También se proporciona un pulsador con el literal «*Default*», cuya función es autocompletar el siguiente bloque con los datos de conexión de las bases de datos de desarrollo que hay en la factoría de software de la empresa, según el servidor de bases de datos seleccionado. Con varias pulsaciones sobre este literal se rota de forma cíclica entre las diferentes conexiones para un mismo sistema de bases de datos, puesto que se da el caso que para algunos sistemas hay varias bases de datos de desarrollo.

Al pulsar el botón «Conectar», se realizan una serie de validaciones mediante JavaScript para comprobar que los datos introducidos tienen el formato correcto:

- *Sistema de bases de datos*, *Host*, *SID* y *usuario* no pueden tomar valores nulos.
- *Puerto* ha de tener valor numérico y no nulo.
- Si no se introduce un valor para el campo *contraseña*, se avisa al usuario mediante un mensaje junto al botón «Conectar», y se hace parpadear brevemente dicho campo para llamar su atención. Si se vuelve a pulsar el botón «Conectar» se lanzará efectivamente la conexión contra el servidor, indiferentemente de que se haya introducido una contraseña.

Si existe algún error en los datos se informará al usuario mediante una caja de texto situada al principio de la página. De lo contrario, se lanzará la conexión al servidor.

Es posible que los datos introducidos, pese a tener un formato válido, no sean correctos y el servidor de bases de datos no responda o rechace la conexión. De ser así, se informará nuevamente al usuario del error y se le permitirá modificar los datos introducidos para volver a lanzar la conexión[1].

7.1.2. Fase 2: Selección de tablas

En la siguiente fase se ofrece una vista que cuenta con una lista doble. En una de las listas aparecen todas las tablas y vistas existentes en la base de datos, que serán añadidas al seleccionarlas a la otra lista. El programador podrá añadir tantas tablas como considere necesarias.

[1]Debido a que algunos sistemas de bases de datos ofrecen más información que otros cuando ocurre un error en la conexión, dependiendo del sistema elegido los textos explicativos del error serán más o menos específicos. Por ejemplo, mientras que un sistema puede devolver un error de tipo *usuario no válido*, otro lanzará simplemente *error en la conexión*.

Al pulsar el botón «Siguiente», se analizan las claves ajenas de las tablas seleccionadas, y se añaden a la lista automáticamente las tablas que son referenciadas por las que el usuario ha seleccionado.

Por ejemplo, si en la aplicación Taller (figura 4.3 en la página 41) se seleccionase la tabla TALLER_RECLAMACION, se añadirían automáticamente las tablas TALLER_REPARACION, TALLER_OPERACION, TALLER_PIEZA, TALLER_VEHICULO y TALLER_PROPIETARIO, conforme al esquema 7.2.

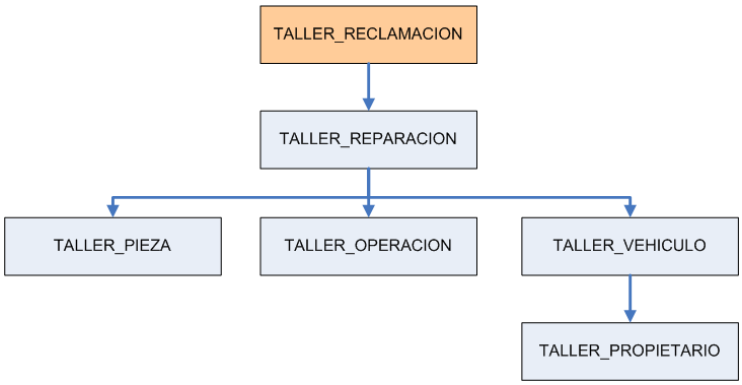

Figura 7.2: Tablas añadidas automáticamente por sus relaciones.

7.1.3. Fase 3: Selección de campos

En esta última etapa se seleccionarán los campos que aparecerán en la interfaz generada por tablas maestras en diversos apartados, tales como el buscador, la vista de edición o la de detalle. Consta de varios bloques.

En el primer bloque se introducirán el nombre de aplicación y el del paquete base.

Los siguientes bloques (tantos como tablas se hayan seleccionado) consisten en un listado de los campos que forman parte de cada tabla, y una serie de opciones que permitirán personalizar la interfaz generada por la aplicación tablas maestras. Dichas opciones son:

Campo por defecto: Para cada entidad se ha de seleccionar un campo por defecto, que será el campo por el que los listados iniciales aparecerán ordenados en la interfaz.

ID: Indica el nombre del campo dentro de la base de datos.

Clase: Indica el tipo de dato SQL de cada campo. Si se sitúa el puntero del ratón sobre este campo, aparece un *tooltip* (mediante el tag tooltip de Genereitor) que indica el tipo de dato Java que le corresponde.

KEY: Indica si un campo es clave primaria de la tabla.

Obl: Indica si el campo tiene la restricción de obligatoriedad activada en la base de datos. De no ser así, se permite establecer dicha restricción a nivel de aplicación. Si se pulsa en el literal de la cabecera de esta columna se activa o desactiva esta opción para todos los campos de la tabla.

Literal: Es el texto que acompañará a las casillas de los formularios generados por la aplicación tablas maestras. Por defecto se ofrece el nombre del campo transcrito a formato Java (ID_PROPIETARIO → idPropietario).

Listado: Permite seleccionar qué campos aparecerán en la vista de listado. Si se activa o desactiva la casilla de la cabecera de la tabla, se activan o desactivan todos los campos de la tabla para dicha opción. Al activarse un campo, aparece un valor en la columna «Idx», que representa el orden de aparición de los campos en la lista. Dicho orden se puede modificar, tarea que se llevará a cabo arrastrando y soltando con el ratón las filas de la columna, hasta ordenar de forma «visual» tal y como se desea que aparezcan los campos. Al soltar los registros los valores del índice que marcan el orden se actualizarán.

Ord: Esta opción sólo está activa para los campos que se han seleccionado para aparecer en el listado, e indica si se ofrecerá la posibilidad de ordenar dicho listado por cada campo en particular. Al seleccionar un campo para listado, se activa esta opción por defecto.

Buscador: Indica los campos que aparecerán en el formulario de buscador.

Idx: Indica la posición en la que aparecerán los campos del formulario de buscador. El orden se establece al seleccionar los campos de forma secuencial, es decir, el primero en seleccionarse recibirá el valor 1, el segundo el 2, etc.

MaxL: Indica la longitud máxima que aceptará la casilla de buscador para cada campo.

Detalle: Indica los campos que aparecerán en el formulario de detalle.

Idx: Indica la posición en la que aparecerán los campos del formulario de detalle. El orden se establece al seleccionar los campos de forma secuencial, es decir, el primero en seleccionarse recibirá el valor 1, el segundo el 2, etc.

MaxL: Indica la longitud máxima que aceptará la casilla de detalle para cada campo.

Claves Ajenas: En esta columna sólo aparecerán opciones para los campos que sean clave ajena de alguna otra tabla. La casilla permite indicar que se desea tener en cuenta la relación entre tablas. Para los campos que formen parte de una misma relación, estas opciones funcionarán de manera paralela, es decir, al activar el usuario un campo, automáticamente se activarán todos los que forman parte de dicha clave ajena.

?: Al poner el cursor del ratón sobre este pequeño literal se mostrará, por medio de un *tooltip*, la tabla y el campo a los que hacen referencia cada clave ajena, a modo informativo.

Campo: Este último seleccionable es una lista de los campos que forman parte de la tabla referenciada por cada clave ajena, permitiendo elegir al usuario el campo de la tabla referenciada que se va a mostrar en la vista generada por tablas maestras. Esto se debe a que generalmente las claves primarias de las tablas son códigos o campos autonuméricos que no aportan un significado claro al usuario, por ello es mejor describir la tabla relacionada por un campo más descriptivo. Por ejemplo, si hubiera una tabla relacionada «persona», sería más intuitivo mostrar el valor del campo «nombre» que el «nif», aunque este último fuera la clave primaria que realmente es referenciada.

También para cada tabla se establece un nombre, que dará título a las vistas de edición, detalle, buscador y edición de cada entidad.

7.2. Resultado

El paquete obtenido al pulsar el botón «generar» contiene los siguientes archivos:

7.2.1. Script para tablas maestras

Es un fichero XML que, en combinación con la aplicación tablas maestras de la empresa, generará la lógica necesaria para tratar las tablas maestras de la aplicación.

Los motivos de utilizar este sistema y no generar un paquete para cada entidad mediante la funcionalidad de *generación de código* son que las tablas maestras son muy sencillas, por lo que con este sistema se reutilizan muchos componentes cuya implementación resultaría redundante si se implementaran de otro modo.

Así, las tablas comunes compartirán la **vista**, puesto que harán uso de los mismos JSP de listado, búsqueda, edición y detalle, tendrán un único componente (EJB o POJO) en la capa de aplicación y un único servlet en la capa web.

El script obtenido para las tablas COMUNIDAD_AUTONOMA, PROVINCIA y MUNICIPIO vistas en la figura 7.1 en la página 74 sería el siguiente:

```
1  <?xml version="1.0" encoding="ISO-8859-15"?>
2  <tablas>
3     <tabla clase="com.comex.aplicacion.dal.bean.ComunidadAutonoma"
           campoPorDefecto="NOMBRE" nombre="Comunidad Autónoma" id="
           APLICACION_COMUNIDAD_AUTONOMA">
4        <campo obligatorio="true" clase="java.lang.Long" pk="true" nombre="
              IdComunidad" id="ID_COMUNIDAD">
5           <detalle maxLength="22" size="100" indice="1"/>
6        </campo>
7        <campo obligatorio="true" clase="java.lang.String" pk="false" nombre="
              Nombre" id="NOMBRE">
8           <listado buscador="true" ordenable="true" size="100" indice="2"/>
9           <buscador maxLength="22" size="100" indice="1"/>
10          <detalle maxLength="22" size="100" indice="2"/>
```

```
11          </campo>
12       </tabla>
13       <tabla clase="com.comex.aplicacion.dal.bean.Provincia" campoPorDefecto="
             NOMBRE" nombre="Provincia" id="APLICACION_PROVINCIA">
14          <campo obligatorio="true" clase="java.lang.Long" pk="true" nombre="
                IdProvincia" id="ID_PROVINCIA">
15             <detalle maxLength="22" size="100" indice="1"/>
16          </campo>
17          <campo obligatorio="true" clase="java.lang.String" pk="false" nombre="
                Nombre" id="NOMBRE">
18             <listado buscador="true" ordenable="true" size="100" indice="1"/>
19             <buscador maxLength="22" size="100" indice="1"/>
20             <detalle maxLength="22" size="100" indice="2"/>
21          </campo>
22          <campo obligatorio="true" clase="java.lang.String" pk="false" nombre="
                Nombre" id="ID_COMUNIDAD_AUTONOMA">
23             <listado buscador="true" ordenable="true" size="100" indice="2"/>
24             <buscador maxLength="22" size="100" indice="2"/>
25             <detalle maxLength="22" size="100" indice="3"/>
26             <fk clase="com.comex.aplicacion.dal.bean.ComunidadAutonoma" tabla="
                   APLICACION_COMUNIDAD_AUTONOMA" valor="ID_COMUNIDAD" campo="
                   FK_TALLER_COMUNIDAD_AUTONOMA_ID_COMUNIDAD">
27                <descripcion>NOMBRE</descripcion>
28             </fk>
29          </campo>
30       </tabla>
31       <tabla clase="com.comex.aplicacion.dal.bean.Municipio" campoPorDefecto="
             NOMBRE" nombre="Municipio" id="APLICACION_MUNICIPIO">
32          <campo obligatorio="true" clase="java.lang.Long" pk="true" nombre="
                IdMunicipio" id="ID_MUNICIPIO">
33             <detalle maxLength="22" size="100" indice="1"/>
34          </campo>
35          <campo obligatorio="true" clase="java.lang.String" pk="false" nombre="
                Nombre" id="NOMBRE">
36             <listado buscador="true" ordenable="true" size="100" indice="1"/>
37             <buscador maxLength="22" size="100" indice="1"/>
38             <detalle maxLength="22" size="100" indice="2"/>
39          </campo>
40          <campo obligatorio="true" clase="java.lang.String" pk="false" nombre="
                Nombre" id="ID_PROVINCIA">
41             <listado buscador="true" ordenable="true" size="100" indice="2"/>
42             <buscador maxLength="22" size="100" indice="2"/>
43             <detalle maxLength="22" size="100" indice="3"/>
44             <fk clase="com.comex.aplicacion.dal.bean.Provincia" tabla="
                   APLICACION_PROVINCIA" valor="ID_PROVINCIA" campo="
                   FK_TALLER_PROVINCIA_ID_PROVINCIA">
45                <descripcion>NOMBRE</descripcion>
46             </fk>
47          </campo>
48    </tablas>
```

Ahora se explicará el significado de cada parámetro de este fichero:

```
1  <tabla clase="com.comex.aplicacion.dal.bean.ComunidadAutonoma"
       campoPorDefecto="NOMBRE" nombre="Comunidad Autónoma" id="
       APLICACION_COMUNIDAD_AUTONOMA">
```

Esta línea indica el comienzo de la descripción de una tabla. Los parámetros que aparecen son los siguientes:

clase indica que las entidades de esta tabla están implementadas en el objeto com.comex.aplicacion.dal.bean.ComunidadAutonoma.

campoPorDefecto indica que la ordenación básica será por el campo NOMBRE.

nombre indica qué literal aparecerá en la aplicación como título de las vistas que hagan referencia a esta entidad. Nótese que hay un espacio y una tilde en la letra «o», lo que indica que dicho campo fue modificado por el programador en el formulario de selección de campos de Genereitor. Este valor es la forma de referirse a la entidad que utilizará la vista.

id indica la tabla de la base de datos que contiene los registros correspondientes a esta entidad.

```
1  <campo obligatorio="true" clase="java.lang.Long" pk="true" nombre="
       IdComunidad" id="ID_COMUNIDAD">
```

Esta línea indica el comienzo de la descripción de un campo de la tabla.

obligatorio indica si el campo ha de tener obligatoriamente un valor o puede aceptar valores nulos.

clase indica el tipo de dato Java que representa a este campo.

pk indica si es clave primaria de la tabla.

nombre indica el nombre por el que será referido en la vista este campo, los encabezados de los listados y los literales que acompañen a las casillas que hagan referencia a esta propiedad.

id indica la propiedad de la entidad de la base de datos a la que se refiere este campo.

```
1  <listado buscador="true" ordenable="true" size="100" indice="2"/>
```

Esta línea indica que el campo aparecerá en las vistas de listado de la entidad.

buscador indica si, además de en el listado, aparecerá en el buscador, ofreciendo la posibilidad de consultar los registros por el valor de este campo.

ordenable indica si se ofrece la posibilidad de ordenar el listado por los valores de este campo.

size indica el tamaño que tendrá la columna del buscador correspondiente a este campo. Este valor tendrá que ser modificado a posteriori por el programador, puesto que es difícil saber el tamaño exacto que corresponderá a cada columna en el momento de la generación.

indice indica la posición en la tabla de listados de este campo.

```
1  <buscador maxLength="22" size="100" indice="1"/>
```

Esta línea indica los parámetros del buscador, en caso de que el campo deba aparecer en dicha vista.

maxLength indica la longitud máxima que aceptará la casilla de búsqueda.

size indica el tamaño que tendrá la casilla. Este valor tendrá que ser modificado a posteriori por el programador, puesto que es difícil saber el tamaño exacto que corresponderá a la casilla dentro del formulario de búsqueda.

indice indica la posición en el formulario de búsqueda de este campo.

```
1 <detalle maxLength="22" size="100" indice="2"/>
```

Esta línea indica los parámetros de la vista detalle, en caso de que el campo deba aparecer en dicha vista.

maxLength indica la longitud máxima que aceptará la casilla de edición/detalle.

size indica el tamaño que tendrá la casilla. Este valor tendrá que ser modificado a posteriori por el programador, puesto que es difícil saber el tamaño exacto que corresponderá a la casilla dentro del formulario de edición/detalle.

indice indica la posición en el formulario de edición/detalle de este campo.

```
1 <fk clase="com.comex.aplicacion.dal.bean.Provincia" tabla="
      APLICACION_PROVINCIA" valor="ID_PROVINCIA" campo="
      FK_TALLER_PROVINCIA_ID_PROVINCIA">
2    <descripcion>NOMBRE</descripcion>
3 </fk>
```

Estas tres líneas indican que un campo es clave ajena que referencia a otra tabla, y que el programado la ha seleccionado para que sea tenida en cuenta.

clase indica el bean que implementa la entidad que es referenciada por la clave ajena.

tabla indica la tabla de la base de datos correspondiente a la entidad referenciada.

valor indica la columna que es clave primaria de la tabla referenciada.

campo indica el nombre que la clave ajena tiene en el sistema de bases de datos.

descripcion indica el campo que aparecerá en el listado de la tabla que estamos tratando referenciando a la relacionada. En este caso, aunque la clave primaria de PROVINCIA sea ID_PROVINCIA, en el listado de MUNICIPIO aparecerá el nombre de la provincia en lugar del identificador, puesto que el nombre ofrece más información al usuario.

7.2.2. Beans

Al igual que en el bloque funcional de *generación de código*, se proporcionan al usuario los beans correspondientes a cada una de las entidades que han sido consideradas *tablas maestras*.

Los beans generados aquí son similares a los que ya se explicaron en la sección de *generación de código* (ver apartado 6.2.8 en la página 71), aunque ofrecen una característica adicional debido a la forma de tratar las relaciones.

Continuando con el ejemplo de COMUNIDAD_AUTONOMA, PROVINCIA y MUNICIPIO (figura 7.1 en la página 74), tomaremos el bean Provincia, ya que posee tanto claves ajenas que referencian a COMUNIDAD_AUTONOMA como referencias desde MUNICIPIO.

De esto se deduce que el objeto Provincia pertenecerá a una ComunidadAutonoma, por lo que poseerá un atributo que almacenará la clave primaria que referencia a dicha entidad. Pero también habrá una serie de objetos de tipo Municipio que harán referencia al objeto Provincia, por lo que el bean correspondiente implementará un nuevo atributo, que representará el listado de objetos Municipio que pertenecen a esta Provincia.

Así, los atributos del bean Provincia serán:

```
1       public static final String TABLA = "APLICACION_PROVINCIA";
2       protected Long idProvincia;
3       protected String nombre;
4       protected ComunidadAutonoma comunidadAutonoma;
5       protected BaseBeanSet listaFkAplicacionMunicipioIdMunicipios;
```

Vemos que, aparte de los atributos propios de la entidad, aparece un atributo del tipo ComunidadAutonoma, que ya aparecía en la *generación de código*, y otro atributo de tipo BaseBeanSet que almacenará el conjunto de elementos de tipo Municipio que pertenecen a la Provincia.

Por último, también se implementan en el bean los métodos get() y set() de todos los atributos, al igual que los beans devueltos con *generación de código*.

Parte IV

Implementación de Genereitor

Genereitor se ha implementado siguiendo las pautas de la especificación J2EE, con una arquitectura de 1+2+1 capas, tal como indica la figura 7.3.

En el capítulo 4 en la página 34 se ha analizado la arquitectura de una aplicación J2EE, por lo que ahora se analizará la forma en la que se ha llevado a cabo la implementación del proyecto.

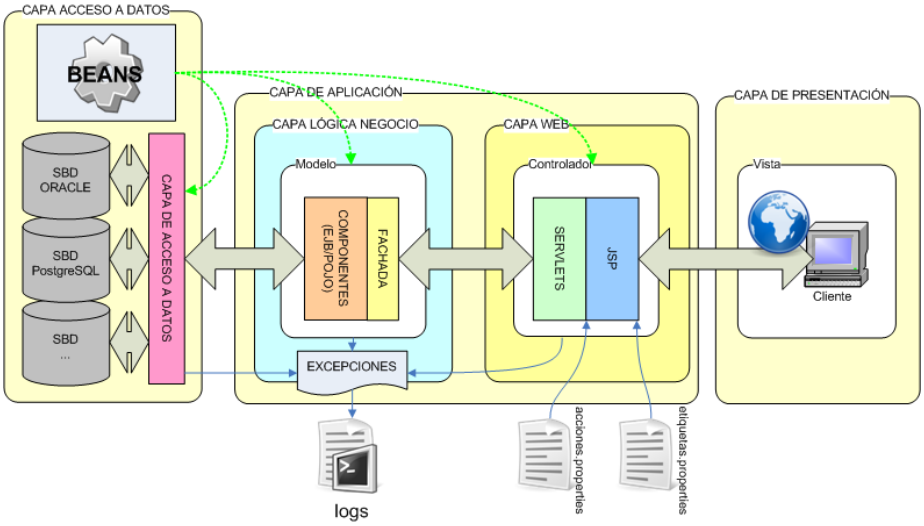

Figura 7.3: Estructura lógica de Genereitor

Capítulo 8

Capa web

La capa de presentación de Genereitor se basa en páginas web HTML generadas dinámicamente con JSP, que incorporan validaciones JavaScript para los formularios de introducción de datos y que comprueban que los datos sean correctos en el lado del cliente, de forma que se evitan conexiones innecesarias o con datos incorrectos al servidor de aplicaciones.

También es una parte fundamental de la capa de presentación el **navegador web**, que cumple la función de interpretar, renderizar y presentar la interfaz de usuario. Usar un navegador web estándar se traduce en un ahorro de tiempo de trabajo, puesto que no es necesario implementar una interfaz completa, y garantiza el acceso a cualquier cliente, puesto que la gran mayoría de sistemas actuales incorpora al menos un navegador en su configuración por defecto.

8.1. JSP

La capa de presentación se ha implementado mediante JSP, que posibilita el uso de Java en páginas HTML para dotarlas de contenido dinámico. Tras la ejecución de un JSP, al usuario (el programador en este caso) se le presenta, vía navegador, una página HTML que representará la interfaz. En C.2 en la página 153 se ofrece una explicación de la tecnología JSP.

Las diversas pantallas que se le presentan al usuario consisten en formularios que éste habrá de rellenar según la acción que desee realizar, enviando los datos al servidor para que, tras ser tratados, se le devuelva, bien a una etapa siguiente del proceso, bien el resultado que desea obtener.

JSP es una tecnología que permite a los desarrolladores y diseñadores web implementar y desarrollar rápida y fácilmente páginas web dinámicas. Estas aplicaciones son multiplataforma. JSP separa el diseño de la interfaz de usuario de la generación de contenidos, lo que posibilita que un diseñador sin conocimientos de la tecnología sea capaz de modificar el aspecto de la página sin alterar los mecanismos de manejo y generación de contenidos dinámicos.

Presenta una serie de ventajas:

- Se puede utilizar la tecnología JSP sin necesidad de aprender Java, utilizando la librería estándar de etiquetas (JSTL). Sin embargo, para aprovechar toda la potencia del sistema es aconsejable poseer conocimientos de Java.
- Es posible extender el lenguaje JSP mediante la implementación de tags personalizados, que amplíen las funcionalidades de la librería estándar.
- Las páginas que se generan son fáciles de mantener, por la mencionada separación entre diseño y generación de contenidos.

La tecnología JSP hace uso de tags al estilo de XML para encapsular la lógica que genera el contenido de la página. Dicha lógica reside en los recursos del servidor (un ejemplo es la arquitectura de componentes JavaBeans), cuyas funcionalidades serán accedidas por la página mediante estos tags o etiquetas.

Esta tecnología es una extensión de los Java Servlets, que son módulos que extienden las funcionalidades de un servidor web. Se descargan bajo demanda a la parte del sistema que los necesita.

JSP y Java Servlets proporcionan independencia de plataforma, rendimiento óptimo, separación de lógica y presentación y facilidad tanto de administración como de uso.

Dado lo complejo de algunos formularios, se ha implementado un control de errores mediante JavaScript, que permitirá al usuario corregir cualquier error en los datos introducidos antes de enviar la petición al servidor. En caso de error, antes de lanzar la conexión al servidor, se informa al usuario de los campos que han de ser corregidos.

También se hace uso de varios *tags*, que facilitan la implementación de las páginas JSP, permitiendo reutilizar código y evitando tener que implementarlos cada vez que se utilicen.

En la figura 8.1 en la página siguiente se presenta una pantalla de la interfaz de Genereitor señalando varios de sus componentes y cómo se han creado. No se señala en el esquema, pero para la apariencia general de la interfaz se han utilizado hojas de estilo CSS.

8.1.1. *Custom Tags*

Los tags son llamadas a objetos predefinidos de la interfaz que se incluyen dentro del cuerpo de una página JSP ofreciendo diferentes funciones, de manera que no es necesario implementarlas cada vez que se utilicen. Son capaces de aceptar parámetros, que serán obligatorios o no dependiendo de la manera en que esté el tag implementado.

La tecnología J2EE permite al programador implementar tags personalizados conforme a sus necesidades, expandiendo las posibilidades del lenguaje HTML.

De esta manera es posible implementar un tag «lista» que incluya en la página generada con el JSP un listado con ciertas características, que recibirá como parámetro el conjunto de datos a listar y otras opciones, de manera que podemos utilizar dicho tag para generar listas en toda nuestra aplicación implementando

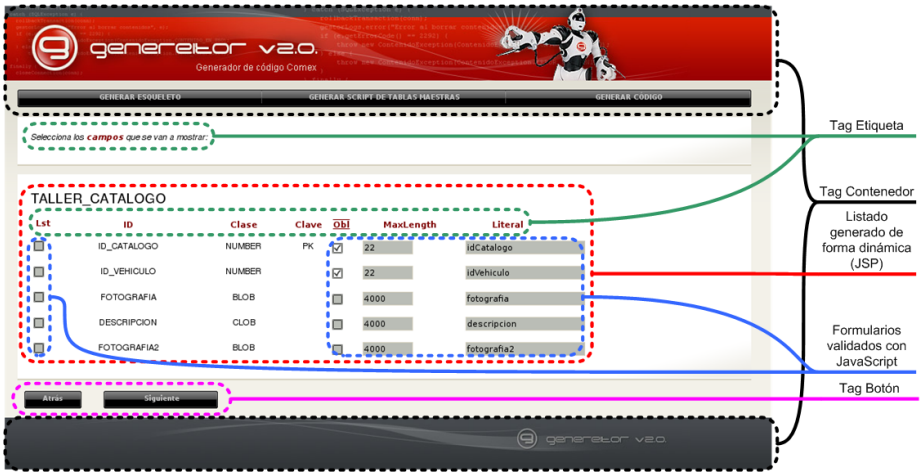

Figura 8.1: Diversas tecnologías utilizadas en la interfaz de Genereitor.

tanto la lógica como la apariencia de dicha lista una sóla vez. En cada página que se utilice la lista, únicamente será necesario incluir su llamada (tag).

Un ejemplo trivial de tag podría ser el siguiente:

Listing 8.1: CabeceraTag.java: tag de ejemplo

```
1  import java.io.*;
2  import javax.servlet.jsp.*;
3  import javax.servlet.jsp.tagext.*;
4
5  public class CabeceraTag extends BodyTagSupport{
6      BodyContent bc;
7
8      public void setBodyContent(BodyContent bc){
9          this.bc = bc;
10     }
11     public int doAfterBody() throws JspException {
12         try {
13             JspWriter out = bc.getEnclosingWriter();
14             out.println("<div align=\"center\"><img src=\"/imagenes/cabecera.
                   jpg\" alt=\"\"></div>");
15             out.println(bc.getString());
16             out.println("<div align=\"center\"><a href=\"mailto:
                   direccion@correo.com\">Contacto</a></div>");
17             bc.clearBody();
18         } catch (IOException e) {
19             system.out.println("Error en Tag Cabecera" + e.getMessage());
20         }
21         return EVAL_BODY_TAG;
22     }
23 }
```

Este sencillo tag incluiría una imagen de cabecera al principio de la página y un link de contacto al final, de forma que se podría utilizar englobando todo el contenido de cada JSP para que todas las páginas compartieran dicha cabecera e información de contacto, sin tener que incluírlo en cada fichero. Un ejemplo de uso sería el siguiente:

Listing 8.2: Página JSP de ejemplo

```
<%@ taglib uri="ejemplos.tld" prefix="ejemplos" %>
<html>
<head>
    <title>Título de la página</title>
</head>
<body>
<ejemplos:cabecera>
...
Contenidos
...
</ejemplos:cabecera>
</body>
</html>
```

Vemos que la primera línea de este último fragmento hace referencia a un fichero `ejemplos.tld`. Dicho fichero contiene las descripciones de todos los tags disponibles en la aplicación, de manera que el servidor de aplicaciones conozca cómo han de ser interpretados y a qué clases corresponden dentro de la implementación de la interfaz.

En dichos descriptores se reflejan los siguientes parámetros:

- Nombre del tag.
- Clase que implementa el tag.
- Atributos aceptados, obligatoriedad de éstos y si son capaces de averiguar sus valores en tiempo de ejecución.
- Información o aclaraciones.
- En caso de que contenga *cuerpo*, se puede indicar a modo informativo qué ha de contener dicho cuerpo.

Un descriptor de tags correspondiente al tag «cabecera» anteriormente descrito será el siguiente:

Listing 8.3: Fichero ejemplos.tld describiendo el tag cabecera

```
<!-- ejemplos.tld -->
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
        "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
   <tag>
      <name>cabecera</name>
      <tagclass>CabeceraTag</tagclass>
      <bodycontent>JSP</bodycontent>
      <info>Incluye imagen de cabecera.</info>
   </tag>
</taglib>
```

8.1.1.1. `<comex:etiqueta>`

Este tag permite el uso de etiquetas en la parte web, cuyo contenido se halla en un fichero etiquetas.properties. El sistema de etiquetas es preferible al

de escribir el texto directamente en la página, puesto que a la hora de hacer modificaciones (e.g. traducir la aplicación) sólo se ha de actualizar el archivo de etiquetas, en lugar de cada una de las páginas.

Uso de EtiquetaTag:

```
<comex:etiqueta clave="clave"/>
```

`clave` (***Obligatorio***) : es el código que identifica a cada etiqueta en el fichero etiquetas.properties.

Es importante destacar que el uso de este tag facilita enormemente la internacionalización de la aplicación: se pueden mantener varios ficheros de etiquetas, uno por cada idioma, y permitir al usuario que seleccione el idioma de su preferencia. De este modo mantenemos la misma implementación de los JSP, siendo el tag el que introduzca los literales correspondientes al idioma escogido.

8.1.1.2. `<comex:listadoble>`

Este tag implementa una lista doble para seleccionar varios campos de una lista extensa de elementos.

Se presentan dos listas, a la izquierda una con todos los campos disponibles, y a la derecha otra con los campos seleccionados. Asímismo se proporcionan dos botones para intercambiar los elementos de una lista a otra.

Al añadir un campo a una lista, se elimina automáticamente de la otra.

Uso de ListaDobleTag:

```
<comex:listadoble id="id" lista="lista" orden=orden/>
```

`id` (***Obligatorio***)**:** el ID que tendrá la lista doble dentro del JSP.

`lista` (***Obligatorio***)**:** el BaseBeanSet que contiene los elementos a listar.

`orden` (***Opcional***)**:** el orden inicial de los elementos en la lista.

Los tres campos soportan expresiones en *request-time*.

Para el uso de este tag se han de indicar dos etiquetas en el etiquetas.properties —correspondientes a los encabezados de cada tabla— cuyas claves serán:

usuario.perfiles.porseleccionar, que corresponde a la tabla de elementos disponibles.

usuario.perfiles.seleccionados, que corresponde a la tabla de elementos seleccionados.

Asímismo requiere dos imágenes, una para cada botón, que han de estar situadas en las rutas:

«`contexto`»`/img/arrow_rigth.gif`, que será el botón para pasar elementos de la lista de elementos disponibles a la de elementos seleccionados.

«`contexto`»`/img/erase.gif`, que será el botón para eliminar elementos de la lista de seleccionados.

8.1.1.3. `<genereitor:tooltip>`

Permite añadir etiquetas informativas sobre campos de texto, especialmente útil cuando no existe mucho espacio disponible en la página (e.g. tablas de gran envergadura) y se han de usar abreviaturas o no es posible mostrar directamente toda la información necesaria.

Uso de TooltipTag:

```
<genereitor:tooltip texto="texto" tooltip="tooltip" long="longitud"/>
```

`texto` **(obligatorio):** es el texto que se mostrará por defecto en la página.

`tooltip` **(opcional):** es el contenido de la etiqueta que se muestra al pasar el puntero sobre el texto. Si no se especifica, se muestra el contenido de `texto`.

`long` **(opcional):** es la cantidad de caracteres de `texto` que se mostrarán en la página. Si el contenido de `texto` es de longitud inferior a este campo y no se ha especificado `tooltip`, no se mostrará etiqueta al pasar el puntero sobre el texto. Si la sobrepasa, cortará el `texto` a los `long` primeros caracteres, seguidos de «...», y se mostrará el contenido entero de `texto` en la etiqueta.

Los tres campos soportan expresiones en *request-time*.

Se puede personalizar la apariencia de la etiqueta definiendo los *style*s siguientes:

- .tooltiptag
- .tooltiptagHover
- .tooltiptag span
- .tooltiptagHover span

8.1.1.4. `<genereitor:contenedor>`

Proporciona el contenedor base para la interfaz gráfica, que será la misma en todos los apartados:

- Fondo.
- Barra de título.
- Barra de menú superior.
- Barra inferior.

Asímismo inicia y termina un formulario, dado que cada página del generador va a constar de un único formulario, de id formulario y método post, cuyos campos irán incluídos entre las etiquetas del tag en cada página.

También proporciona una caja de contenido para mostrar errores de conexión, validación de datos, información adicional o advertencias.

Para mostrar u ocultar esta ventana existen dos funciones Javascript:

`showbox("tipo",mensaje);`, que al ejecutarse hará visible la caja de información, mostrando un icono según el primer parámetro (cuyos valores pueden ser «`info`» y «`error`») y el mensaje. Si el primer parámetro no coincide con los valores señalados, no se muestra ningún icono. Tras esto, va al inicio de la página.

`hidebox();` Oculta la caja de información.

En la barra de menú superior se ubican los enlaces a las cuatro partes del generador: generación de esqueleto, de script de tablas maestras, de código y ajustes. Para cada uno de estos cuatro enlaces, se requieren las siguientes etiquetas en el etiquetas.properties:

- boton.esqueleto
- boton.script
- boton.codigo

También necesita las siguientes imágenes:

«`contexto`»`/imagenes/img_pie.jpg`: imagen del pie de página.

«`contexto`»`/imagenes/icono_ok.gif`: icono de confirmación para la infobox.

«`contexto`»`/imagenes/icono_error.gif`: icono de error para la infobox.

«`contexto`»`/imagenes/g.ico`: icono que aparecerá en la barra de direcciones (*favicon*[1]), con el logotipo de Genereitor.

[1] *Favicon* es como se conoce al icono que aparece en la barra de direcciones de la mayoría de navegadores, junto a la URL de la página visitada, y en la lista de favoritos en caso de añadir tal dirección.

Y los siguientes estilos:

#contenedor_general, que define el estilo general del cuerpo de la página (`fnd_body.gif`).

#cabecera, que define la cabecera de la página (`img_cabecera.jpg`).

#contenedor, que define el estilo del contenedor de la página.

#navegador_top, que define el estilo de la barra de menú superior (`fnd_navtop.gif`).

- .opcion_navtop, estilo de cada opción del menú superior.
- .navtop, aspecto de los botones del menú superior.
- .navtop:hover, aspecto de los botones del menú superior al pasar el puntero sobre ellos.

#pie, que define el estilo del pie de página (`fnd_pie.gif`).

Uso de ContenedorTag:

```
<genereitor:contenedor accion="accion">
⋮
</genereitor:contenedor>
```

- `accion` *(Opcional)*: define la acción del formulario.

8.1.1.5. `<genereitor:boton>`

Tag que permite la inclusión de botones en la web.

Uso de BotonTag:

```
<genereitor:boton valor="valor" ancho=ancho/>
```

- `valor` (*Obligatorio*): define el texto que llevará el botón.
- `id` (*Opcional*): define el id del botón dentro de la página.
- `javascript` (*Opcional*): define una función Javascript asociada al botón.
- `onclick` (*Opcional*): define un evento al hacer click sobre el botón.
- `href` (*Opcional*): define un enlace para el botón.
- `orden` (*Opcional*): define el orden del botón en la página.

- `ancho` (*Opcional*): define el ancho del botón (1, 2 ó 3).
- `css` (*Opcional*): permite establecer un estilo para el botón.
- `enabled` (*Opcional*): permite activar o desactivar el botón.
- `title` (*Opcional*): define el título del botón.

Todos los campos soportan expresiones en *request-time* excepto id.

8.1.2. JavaScript

Parte de la interfaz gráfica de Genereitor consta de varios formularios generados de forma dinámica conforme a los metadatos de las tablas de la base de datos contra la que se está trabajando. Según las características de dichas tablas, es común que se presenten al usuario formularios con gran cantidad de campos para rellenar o verificar.

Dada esta complejidad, no es raro que el usuario cometa fallos a la hora de rellenar los formularios, olvide introducir opciones o las rellene de forma incorrecta. Por esto, se implementa un control de errores que verifica que se den las siguientes condiciones:

- En la etapa de conexión:
  - Los campos *host*, *SID* y *usuario* no se encuentren vacíos.
  - El campo *puerto* no se encuentre vacío y sea un campo numérico válido.
  - El campo *contraseña*, en caso de estar vacío, se advertirá al usuario mediante un mensaje junto al botón de conectar de tal situación, y el campo *contraseña* parpadeará brevemente. Si se vuelve a pulsar el botón conectar, se enviará la conexión al servidor con la contraseña vacía, puesto que es posible que exista acceso sin contraseña.
- En los formularios de selección de campos de tabla, que haya por lo menos un campo señalado como campo de ordenación por defecto para cada tabla.
- En el formulario de selección de campos de tabla del bloque funcional de generación de script de tablas maestras, se comprobará que haya por lo menos un campo para buscador, un campo para detalle y un campo para listado en cada tabla.
- Que ningún campo de nombre de bean, alias de campo, nombre de aplicación, paquete base o longitud máxima de campo esté vacío.
- Que todos los campos de longitud máxima sean de tipo numérico.

En caso de incumplirse alguna de las anteriores condiciones, se retorna al comienzo de la página (en caso de que la misma supere la altura de la ventana del navegador), y se muestra una cabecera informando de cada error sucedido, de forma que sean fácilmente localizables para el usuario. Los datos introducidos hasta el momento se mantienen, ofreciendo al programador la posibilidad de corregir los errores y volver a confirmar el formulario.

Una vez se hayan superado las validaciones JavaScript se procederá a lanzar la conexión al servidor de aplicaciones, que recogerá toda la información introducida en el formulario.

El motivo del uso de JavaScript para realizar dichas validaciones es precisamente la complejidad de los formularios. Si se tuviera que comprobar la validez de los datos introducidos en el servidor de aplicaciones, se perdería mucho tiempo recargando la interfaz en cada verificación, puesto que se tendrían que enviar al servidor los datos, comprobarlos y devolverlos a la interfaz en caso de error. Usando JavaScript se descarga la red y el servidor, asegurando que los datos que envía el usuario son válidos.

8.1.3. CSS

Cascading Style Sheets (hojas de estilo en cascada) es un lenguaje utilizado para definir la presentación de un documento escrito en HTML o XML. Esto establece una separación completa entre contenido y presentación. Con CSS se definen colores, tipografías, capas, márgenes y otros aspectos de la presentación del documento.

Esta separación proporciona una ventaja en cuanto a la accesibilidad de los contenidos. Por ejemplo, en un sitio web podríamos definir tres hojas de estilo diferentes:

- Una con los colores corporativos de la empresa, que resultase atractivo visualmente.
- Una con fondos blancos y textos en negro, y ocultando elementos tales como menús o elementos al márgen, para lograr copias impresas de calidad.
- Una con tipografías grandes y colores de alto contraste, para facilitar el acceso a los contenidos a personas con dificultades o problemas visuales.

Así, las ventajas que ofrece son:

- Centraliza el control de la presentación de un sitio web completo con un sólo fichero de estilos, lo que mantiene la concordancia entre la apariencia de los diferentes apartados del mismo.
- Es posible que el usuario especifique su propia hoja de estilo para un sitio web, de forma que pueda adecuarla a sus preferencias o necesidades de accesibilidad.
- Se pueden definir varias hojas de estilo diferentes, según el dispositivo mostrado (ordenador, pda, página para imprimir).

- El código HTML es más claro de entender y se reduce su tamaño, al externalizar las definiciones de estilo.

Por último, conviene destacar que CSS es un estándar de W3C. Tanto Genereitor como las interfaces que genera utilizan hojas de estilo correspondientes a la versión 2.1[2] de CSS.

8.2. Servlets

Bajo JSP y las diferentes tecnologías que lo complementan, explicadas en el apartado anterior, se encuentran los *servlets*, y que dentro del patrón estructural de modelo-vista-controlador realizan la función de controlador.

Los servlets hacen de intermediarios entre la vista y el modelo, al ser más versátiles que JSP para esta función al estar escritos como clases Java normales, evitando de esta forma mezclar código visual (HTML, XML) con código Java, delegando a JSP únicamente la función de componer la interfaz visual. Cada vínculo en la interfaz web se corresponde con una acción de un servlet.

Los servlets tienen una función perform(), que es la encargada de recibir la petición del usuario que llega de la interfaz, y dirigir la ejecución hacia la función correspondiente, tal como muestra la figura 8.2.

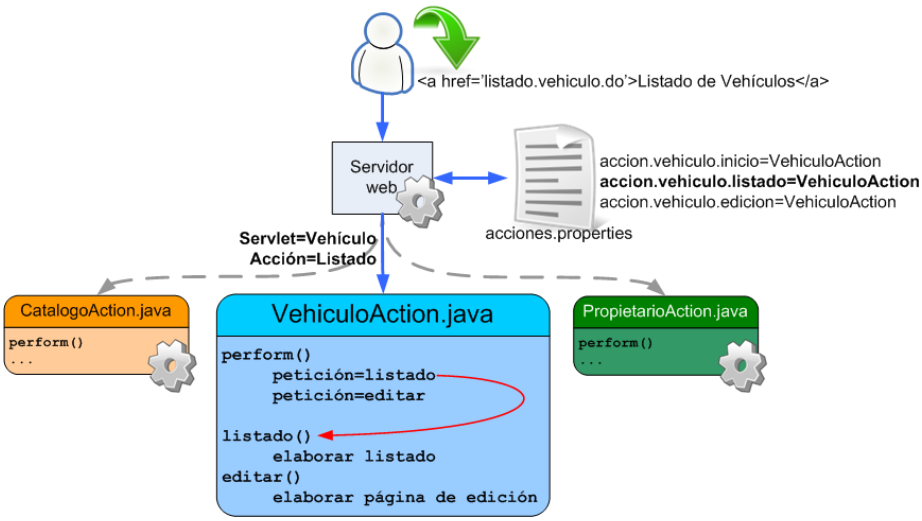

Figura 8.2: Reconocimiento de las peticiones del usuario por el controlador.

El controlador de Genereitor se compone de seis servlets, que realizan funciones específicas:

[2]La especificación de CSS 2.1 puede ser consultada en http://www.w3.org/TR/CSS21/.

8.2.1. InicioAction.java

Este primer servlet es el más sencillo de todos, y es el encargado de cargar la página de bienvenida a Genereitor, que dará acceso a las secciones correspondientes a los tres bloques funcionales, los cuales cuentan con un servlet dedicado para cada uno.

Únicamente cuenta con dos funciones: perform, que será la encargada de consultar la petición del usuario que le llega desde la vista y dirigirla hacia la función correspondiente, e inicio, que llama a la vista para que muestre la interfaz correspondiente, en este caso la resultante de compilar el fichero index2.jsp[3].

8.2.2. GenereitorBaseAction.java

Es el servlet «padre» de TablasAction, EsqueletoAction y CodigoAction, de manera que los tres hereden los métodos que implementa.

Este servlet contiene funciones comunes a varios bloques funcionales:

Paginador. Para cada vista generada, se requiere un paginador, que es el encargado de dar forma a las listas que se mostrarán.

Operación de conexión. Esta operación es común a los tres bloques funcionales[4], por lo que se implementa en este servlet para no repetir dicha función en los servlets correspondientes a cada bloque.

Recolección de datos de conexión: Se encarga de recoger los datos de la conexión de los formularios. Dado que dichos datos son comunes a las tres, la función encargada de dicha tarea se implementa.

Compresión: También aquí se implementa la función que crea en el servidor el fichero comprimido que se devolverá al usuario al final de cada ejecución de todos los bloques funcionales.

Llamadas a la lógica de la aplicación: Son las funciones que recolectan los datos devueltos por la lógica de la aplicación:

- Lista de tablas de la base de datos.
- Lista de columnas de una tabla.
- Lista de claves ajenas de una tabla.
- Lista de claves primarias de una tabla que son ajenas de cualquier otra.

[3] Al iniciar una conexión contra un servidor web, siempre se busca el fichero index.htm, index.php o similar. En este caso existe un index.jsp, que únicamente contiene una redirección hacia la acción inicio.inicio.do, que se encargará de llamar al controlador para que genere la vista inicial, contenida en el fichero index2.jsp.

[4] Si bien sólo es necesaria una conexión con la base de datos en los bloques de generación de código y script de tablas maestras, recordemos que se incluyó la posibilidad de comprobar la conexión en la etapa de generación de esqueleto, a fin de verificar que los datos introducidos son correctos.

Recolección de datos de formularios: Funciones que recolectan los datos introducidos en los formularios de la aplicación y componen los nodos XML de datos que luego serán combinados con las plantillas XSL para generar los ficheros que se devolverán posteriormente al usuario.

Creación y borrado de temporales: Tanto los ficheros generados como los ficheros que se contienen se almacenan durante la sesión en el servidor. Estas funciones se encargan tanto de crearlos como de borrarlos cuando ya no son necesarios.

8.2.3. EsqueletoAction.java

Dado que el bloque funcional de generación de esqueleto sólo se compone de una etapa, el servlet correspondiente dispone únicamente de dos funciones de acceso público.

La primera se encargará de formar la vista que presentará al usuario el formulario de introducción de los datos requeridos.

La segunda encarga de recolectar los datos específicos del formulario de la interfaz, combinar dichos datos, una vez formateados con XML con las plantillas XSL, copiar los ficheros de uso común (imágenes, hojas de estilo, páginas de error genéricas), empaquetarlos y devolverlos al usuario.

Para esto, el servlet llama a su clase superior, GenereitorBaseAction, aprovechando las funciones que implementa.

También subdivide el proceso en funciones recursivas para optimizar el código.

Por último, implementa una función *conectar*, que es simplemente una llamada a la función homónima de GenereitorBaseAction.

8.2.4. CodigoAction.java

Este servlet implementa acciones para cada etapa de la que consta el bloque funcional de generación de código.

8.2.4.1. Inicio

Al igual que el anterior, la primera etapa cuenta con una función de inicio, encargada de dibujar la interfaz de introducción de los datos de conexión.

8.2.4.2. Conexión a la base de datos

Esta función es una mera llamada a la función homónima de la clase superior, GenereitorBaseAction, que lanza la conexión a la lógica de la aplicación.

8.2.4.3. Selección de tabla

Tras efectuar la conexión, la segunda opción consulta a la fachada de la lógica de negocio para obtener un listado de todas las tablas existentes en la base de datos, que se pasarán a la vista como parámetro de un tag *lista*, y se evaluará si dicha tabla posee alguna clave ajena.

Si se da el caso de que existen tablas cuya clave primaria sea una clave ajena de la tabla seleccionada, se presentará al usuario una lista de las tablas relacionadas. Si no, este paso se omite.

8.2.4.4. Selección de tablas relacionadas

Si es necesario, se crea una vista que muestra al usuario una lista doble, para que seleccione las tablas relacionadas que se quiere tener en cuenta. Como ya se ha dicho, en caso de que la tabla principal no posea relaciones, este paso se omite.

Tanto la tabla principal como las relacionadas si las hubiera, se almacenan como parámetros de sesión, para su recuperación posterior a la hora de generar el paquete.

8.2.4.5. Selección de campos

Tras la selección de tablas, se consulta a la lógica de la aplicación los campos de cada tabla que ha sido seleccionada, mostrándose una vista que permite al usuario establecer los campos que desea que aparezcan en los listados de la aplicación que se va a generar, así como las restricciones de obligatoriedad y longitud máxima que aceptará. También se permite modificar el nombre que cada propiedad de las entidades de la tabla recibirá al ser implementados como objetos Java.

Dicho listado será generado dinámicamente con JSP.

Tras confirmar que los datos introducidos son correctos, se almacenarán como parámetros de sesión y se presentará al usuario la vista correspondiente a la última etapa del proceso.

8.2.4.6. Selección de parámetros y generación de paquete

En esta fase se presenta nuevamente un formulario al usuario. Al pulsar el botón «Generar» se llama a la función homónima de este servlet, que se encargará de recolectar los parámetros de este último formulario, así como los atributos almacenados en la sesión, para luego llamar a la función de GenereitorBaseAction que convertirá dichos parámetros en nodos de datos XML. Tras esto, creará la estructura de directorios necesaria para organizar los ficheros que formarán parte del paquete que se devolverá al usuario y se combinarán, según los parámetros introducidos, las plantillas XSL necesarias para generar dicho paquete. Tras esto, se llama a la función «comprimir» de la clase superior, y se devuelve el paquete al usuario.

Esta última etapa está descompuesta en varias funciones para hacerla más modular, de forma que el mantenimiento de la aplicación sea más sencillo.

8.2.5. TablasAction.java

El servlet correspondiente al bloque funcional de tablas maestras tiene cinco funciones principales:

8.2.5.1. Inicio

Al igual que el anterior, la primera etapa cuenta con una función de inicio, encargada de dibujar la interfaz de introducción de los datos de conexión.

8.2.5.2. Conexión a la base de datos

Esta función llama a la función homónima de la clase superior, GenereitorBaseAction, que lanza la conexión a la lógica de la aplicación. Tras esto, presenta al usuario la vista de selección de tablas.

8.2.5.3. Selección de tablas

Tras recuperar las tablas que ha seleccionado el usuario, se añaden a la lista automáticamente todas las tablas referenciadas por las claves ajenas de las seleccionadas, y se elabora la vista de selección de campos y las propiedades de éstos, que se presentará al usuario.

8.2.5.4. Selección de campos y generación de paquete

Tras recopilar la información del formulario que acaba de rellenar el usuario, se procede a elaborar los nodos XML que reunen toda la información necesaria, y se combinan con las plantillas XSL oportunas para generar el paquete que contendrá el script de tablas maestras y los beans correspondientes a las entidades para las que se ha generado dicho script.

8.2.6. ServletEscuchador.java

Este último servlet es el encargado de borrar los ficheros temporales que ya no se usan.

Al generar un paquete, se borra inmediatamente la estructura de directorios que se había generado en el servidor de aplicaciones para confeccionarlo, pero dicho paquete comprimido todavía no se ha borrado, puesto que la aplicación no es capaz de saber si el usuario ha aceptado o no la descarga. Si se eliminase el paquete comprimido antes de que se hubiera completado la descarga, no se acabaría de transferir completamente.

Por esto, es éste servlet el encargado de hacer esta «limpieza», vigilando el tiempo de vida de las sesiones iniciadas en la aplicación, y al momento de la muerte de cada sesión, elimina los paquetes comprimidos correspondientes del servidor de aplicaciones. De este modo no quedan *residuos* que, de otro modo, acabarían por llenar la capacidad de almacenamiento del servidor que soporta Genereitor.

8.3. Otros

Aparte de los ficheros mencionados hasta ahora, también forman parte de la capa web diversos componentes que ofrecen varias utilidades:

UtilesWEB.java: Este fichero implementa una serie de utilidades estándar que se usarán en los servlets, tales como validación de campos numéricos, formateo de fechas, funciones conversoras de HTML a texto plano, comprobación de fechas, etc.

ConstantesWEB.java: Aquí se recopilan diversos valores constantes que se utilizan en la aplicación, como por ejemplo nombres de atributos de sesión. Sólo se hallan aquí las constantes de propósito general, las específicas de cada bloque funcional se declaran en el servlet correspondiente.

Capítulo 9

Capa de lógica de negocio

La lógica de negocio se implementa en Genereitor mediante POJOs (*Plain Old Java Objects*), que son clases normales de Java. Hemos visto al analizar el funcionamiento de la arquitectura J2EE que ésta proporciona la tecnología EJB para implementar la capa de lógica de negocio. No obstante, para esta capa Genereitor no requiere de la potencia ni características de dicha tecnología, por lo que utilizando clases *Java* lograremos un diseño más sencillo, a la vez que ahorraremos en recursos ya que no se requerirá de un contenedor de EJBs para ejecutarlo.

Así, el único componente que formará la capa de lógica de negocio será Conexion.

Esto es debido a que realmente Genereitor no maneja datos, para la generación de una aplicación web es indiferente que la base de datos contra la que se trabaja contenga o no datos.

Por contra, esta sencillez del diseño se ve truncada al enfrentarnos a la necesidad de conectar a varios sistemas de bases de datos, por lo que tenemos que es necesario realizar una implementación de la lógica de negocio para cada conexión. No obstante, todas las implementaciones compartirán fachada, de forma que esta diferencia sea transparente en cuanto a los componentes de las capas superiores.

Así, la lógica de negocio, compuesta por el objeto Conexión, tiene las siguientes funciones:

conectar() lanza una conexión a la base de datos a partir de un filtro de tipo conexion. Su única función es comprobar que la conexión es correcta y que el servidor de bases de datos responde.

getTablas() recupera las tablas de la base de datos, a partir de un objeto conexion.

getColumnas() recupera las columnas de una tabla, a partir de un objeto conexion y un filtro tabla.

getClavesExportadas() devuelve información sobre las tablas que tienen por clave primaria un campo que es clave ajena de la tabla sobre la que se trabaja.

getClavesImportadas() devuelve información sobre las tablas que tienen por clave ajena la clave primaria de la tabla sobre la que se trabaja.

Es importante señalar que normalmente las aplicaciones que trabajan sobre una base de datos almacenan los datos de conexión en un fichero de *datasources*, que almacena todos los datos de conexión.

Por la naturaleza de Genereitor, esto sería imposible, ya que es necesario que sea capaz de conectarse a cualquier base de datos, cuyos parámetros de conexión serán introducidos por el usuario. Así, el tipo de objeto Conexion se utiliza para almacenar estos datos.

9.1. Excepciones

La clase GenereitorException reúne los códigos de excepción que Genereitor manejará en caso de producirse cualquier error. A fin de proporcionar la información precisa de la causa de cada error, se definen una serie de códigos de excepción, que equivalen a diversos mensajes que se reflejarán en el sistema de gestión de *logs*:

```
1       public static final int ERROR_GENERAL = 0;
2       public static final int CONEXION_RECHAZADA =4;
3       public static final int DATOS_NO_VALIDOS = 5;
4       public static final int USER_NO_VALIDO = 50;
5       public static final int PASS_NO_VALIDO = 51;
6       public static final int TABLA_NO_ENCONTRADA = 6;
7       public static final int SID_NO_VALIDO = 7;
8       public static final int CLASS_NOT_FOUND = 999;
```

Hay excepciones que pueden resultar redundantes (e.g. datos no válidos, user no válido, pass no válido). Esto es debido a que algunos sistemas de bases de datos hacen distinciones al lanzar una excepción, haciéndolo de manera más específica, mientras que otros informan de los errores de forma más general. Los códigos de error que lanzan cada base de datos se consultan en los componentes de la lógica de negocio de cada sistema, lanzando una GenereitorException acorde al código de la SQLException generada por el sistema de bases de datos.

Capítulo 10

Capa de acceso a datos

10.1. Componentes de acceso a datos

Estas clases son las encargadas de comunicarse con el sistema de bases de datos. Nuevamente se plantea el problema de la necesidad de conexión a diferentes sistemas de bases de datos, por lo que se han de realizar implementaciones diferentes para cada uno.

No obstante, todas las implementaciones ofrecen las mismas funciones:

conectar() se utiliza simplemente para comprobar la conexión con el sistema de bases de datos.

getTablas() devuelve las tablas de la base de datos.

getColumnas() devuelve las columnas de una tabla de la base de datos.

getPrimaryKeys() devuelve los campos que son clave primaria de una tabla de una base de datos.

Para recuperar los metadatos de las bases de datos se hace uso de la clase java.sql.DatabaseMetaData.

10.2. Sistema de base de datos

Dado que Genereitor se puede conectar a cualquier base de datos –mientras esté soportado el sistema de bases de datos– no se puede realizar un análisis del funcionamiento de dicha base de datos.

10.3. Beans

Los beans son un modelo de componentes de Sun que permite encapsular varios objetos en uno único[1]. De este modo se consigue un desarrollo más intuitivo, al usar objetos «completos» en lugar de conjuntos de objetos más simples. Realmente no pertenecen a una capa específica, ya que están disponibles a todos los niveles de la aplicación.

En Genereitor existen cuatro tipos de Bean:

BD: representa un objeto «base de datos».

Tabla: representa un objeto «tabla».

Columna: representa un objeto «columna».

Conexion: representa un objeto «conexión».

Para explicar el funcionamiento de un bean, se presenta el ejemplo más sencillo de los utilizados.

El bean conexion representa un objeto «conexión» cuyos atributos corresponden a los que se introducen en las etapas de conexión a base de datos de Genereitor.

```
1  public class Conexion extends BaseBean implements Serializable {
2      private String host;
3      private String password;
4      private Integer port;
5      private String sid;
6      private String tipo;
7      private String user;
8
9      public Conexion() {
10         host = null;
11         port = null;
12         sid = null;
13         user = null;
14         password = null;
15         tipo = null;
16     }
17     public String getHost() {
18         return host;
19     }
20     public String getPassword() {
21         return password;
22     }
23     public int getPort() {
24         if (port != null) {
25             return port.intValue();
26         } else {
27             return 0;
28         }
29     }
30     public String getSid() {
31         return sid;
32     }
33     public String getTipo() {
34         return tipo;
35     }
```

[1]La especificación de JavaBeans se puede consultar en http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp.

```
36      public String getUser() {
37          return user;
38      }
39      public void setHost(String host) {
40          this.host = host;
41      }
42      public void setPassword(String password) {
43          this.password = password;
44      }
45      public void setPort(int port) {
46          this.port = new Integer(port);
47      }
48      public void setSid(String sid) {
49          this.sid = sid;
50      }
51      public void setTipo(String tipo) {
52          this.tipo = tipo;
53      }
54      public void setUser(String user) {
55          this.user = user;
56      }
57  }
```

Como se aprecia, existe un constructor vacío del objeto, que creará el objeto pero sin valores. Luego, mediante las propiedades get y set se establecen los diferentes atributos. Para el ejemplo se ha simplificado el bean, pero realmente por cada propiedad existen cuatro métodos:

- getPropiedad
- getPropiedadObject
- setPropiedad
- setPropiedadObject

Las propiedades Object son necesarias para cuando para un tipo de datos existe tipo primario y tipo objeto, por ejemplo int e Integer. En este caso, la función Object podrá adoptar o devolver valores nulos, mientras que la función primaria devolvería 0 en su lugar[2].

Nuevamente, al ser el tratamiento de los tipos de datos diferentes[3] para cada base de datos, los beans Tabla y Columna se han de implementar una vez por cada sistema de bases de datos soportado. No obstante, se mantiene una especificación común a todos, a modo de fachada, de manera que para el resto de componentes de la aplicación el acceso sea transparente.

[2]El comportamiento se implementará según convenga.

[3]Un ejemplo de esto es el uso de oracle.jdbc.driver.OracleTypes para bases de datos Oracle, mientras que para el resto se utiliza java.sql.Types.

Parte V

Conclusión

Capítulo 11

Visión global del trabajo realizado

Con Genereitor se ha conseguido elaborar una herramienta de generación de código funcional y personalizada a los requerimientos de cada proyecto, ofreciendo al programador la posibilidad de comenzar el desarrollo del mismo a partir de una aplicación que ya funciona, y que dispone de métodos estándar a todos los niveles. Esta tarea, que sin la ayuda de Genereitor cuesta varios días, es resuelta ahora en cuestión de 5 minutos, ofreciendo además una implementación de calidad, respetando estándares y convenciones del lenguaje.

Para la implementación de Genereitor se han utilizado tecnologías tales como J2EE, puesto que proveen de una estructura modular que facilita el desarrollo y el mantenimiento de la aplicación, así como la adición de nuevos módulos y componentes.

La elección del desarrollo de la aplicación en Java/J2EE ha sido tomada por diferentes motivos:

- Es una tecnología muy utilizada para el desarrollo de aplicaciones web, por lo que existe un gran soporte y mucha documentación disponible.
- Tiene mucho tiempo de experiencia en el sector, por lo que está muy desarrollada, lo que conlleva el desarrollo de aplicaciones maduras y estables.
- Su aprendizaje no es excesivamente difícil, por lo que se prefiere a otras tecnologías más complicadas.
- Sirve como experiencia para el desarrollo de las plantillas de las aplicaciones que se generarán con Genereitor, que obligatoriamente han de estar desarrolladas en esta tecnología.
- Se ha optado por utilizar en la medida de lo posible software libre y gratuíto, por lo que el uso de J2EE se prefiere a tecnologías propietarias o con

licencias comerciales, tales como .Net. Por el mismo motivo, el servidor de aplicaciones sobre el que se desplegará Genereitor es Apache Tomcat[1]

La entrada en producción de Genereitor se ha llevado a cabo de forma progresiva, poniendo a disposición de los programadores primero el módulo de *generación de script de tablas maestras* en febrero, y los módulos de *generación de esqueleto* y *generación de código* en abril.

El proyecto actualmente tiene aproximadamente 1500 horas de desarrollo. En este tiempo van incluídos los periodos de formación en las tecnologías utilizadas, y la elaboración del documento de análisis y diseño previo a la implementación de la aplicación (disponible en el apéndice A en la página 116).

11.1. Cambios en Genereitor durante su desarrollo

Aunque se elaboró un documento de análisis y diseño donde se redactaron al detalle las características que tendría Genereitor, durante la etapa de implementación y pruebas se llevaron a cabo diversos cambios, bien debidos a modificaciones en los requisitos generales de las aplicaciones, bien debido a errores o mejoras descubiertos durante las pruebas[2]. Algunas de estas modificaciones fueron:

Relaciones entre tablas maestras

En un principio sólamente se iba a tener en cuenta un nivel de relaciones entre las tablas para la generación del script de tablas maestras, es decir, no se tendrían en cuenta las tablas *relacionadas de las relacionadas*. Además, se iba a presentar al usuario una lista para la selección de las que considerase oportunas.

Pero una vez implementada la lógica que extraía las relaciones de la base de datos, se llegó a la decisión de que era interesante tener en cuenta las relaciones entre las tablas a todos los niveles, puesto que es común que las tablas maestras tengan varios niveles de relaciones. Un par de ejemplos que ilustran dicha situación:

- País, Comunidad autónoma, Provincia, Comarca, Municipio...
- Eón, era, período, época, edad...

Una era contiene varios períodos, que a su vez contienen varias épocas, etc.

Además, se decidió eliminar la posibilidad de elección del usuario, agregando automáticamente todas las tablas relacionadas, porque comúnmente se seleccionaban todas las tablas del *árbol de relaciones*, así que seleccionándolas todas

[1] Con licencia Apache v2.0, cuyos términos se pueden consultar en http://www.apache.org/licenses/GPL-compatibility.html.

[2] Cuando el desarrollo de Genereitor estaba avanzado, se abrió el acceso a los programadores del departamento de J2EE de la empresa, cuyo *feedback* aportó nuevas ideas y propuestas de soluciones a los problemas.

automáticamente eliminamos la posibilidad de que el usuario por error olvide seleccionar una tabla.

Beans en tablas maestras

Los beans son generados en la etapa de *generación de código*. No obstante, dicho bloque funcional sólo tiene en cuenta automáticamente las relaciones en un sentido, mientras que *tablas maestras* trabaja con relaciones en ambos sentidos (importadas y exportadas). Es por ello que se decidió proporcionar también los beans de las tablas maestras en esta bloque funcional, de forma que se doten automáticamente de los atributos correspondientes a las relaciones de ambos sentidos.

En el ejemplo anterior, de las entidades eón, era, período, época y edad, un período pertenece a una era, por lo que el bean correspondiente poseerá un atributo *era*. Este atributo también es añadido por *generación de código*.

Pero al mismo tiempo, un período tiene una serie de épocas, por lo que se le añadirá un atributo *listaEpocas*.

PL/SQL

En principio no se contempló la posibilidad de generar código PL/SQL, pero tras comprobar que la mayoría de los proyectos usaban esta tecnología, se decidió dar soporte en la capa de acceso a datos, y proporcionar las funciones PL/SQL que implementarán la capa de acceso a datos en el propio servidor de bases de datos, liberando la carga de trabajo tanto al servidor de aplicaciones como a la red.

Módulos web

La opción en *generación de esqueleto* y *generación de código* que permite introducir varios módulos web al principio no existía. Se generaba automáticamente el módulo *aplicación*-web y si el programador seleccionaba una casilla de «generar módulo de administración», se generaba también el módulo *aplicación*-webadmin. Pero al final se decidió dar opción de personalizar el nombre y el número de los módulos web.

Listado de entidades relacionadas en la vista de edición de la principal

En principio se generarían vistas de edición y detalle de manera independiente para todas las tablas, tanto la principal como las relacionadas. Luego se decidió añadir la opción de incluir el listado de las tablas relacionadas en la pantalla de edición de la entidad principal, puesto que esta vista era usada con frecuencia en las aplicaciones desarrolladas.

Streaming

Debido a un *bug* en OC4J que provoca que al recibir archivos binarios de cierta envergadura desde un formulario el servidor de aplicaciones colapse por desbordamiento de memoria, se decidió implementar las operaciones con binarios por trozos, tanto para la subida al servidor de bases de datos como para la descarga. De esta manera se evita el error en caso de usar OC4J, se mejora el rendimiento de la aplicación y permite que, en caso de utilizarse reproductores de medios (audio, vídeo), el tiempo de carga del *buffer* sea menor, al tener que descargar sólamente una parte del archivo y comenzar a reproducirla mientras se descargan las demás, en lugar de traer el fichero completo y entonces comenzar la reproducción.

11.2. Conocimientos adquiridos

Con la elaboración de Genereitor se han adquirido gran cantidad de conocimientos, tanto relativos a tecnologías como a metodologías de trabajo. Algunas de estas son:

Java: se ha adquirido gran experiencia en este lenguaje, del que al principio no se tenía apenas conocimientos.

J2EE: se ha familiarizado con esta tecnología para el desarrollo de aplicaciones web, así como de las APIs que implementa, tales como conectores a bases de datos (jdbc), JavaServlets, JSP, gestión transaccional, etc. También se han adquirido conocimientos de aplicaciones desarrolladas en capas, la división de tareas y modularidad de las aplicaciones.

Patrones de diseño de aplicaciones: tales como el Modelo-Vista-Controlador, Facade, Filtro, Singleton o ServiceLocator.

Tecnologías usadas en la web: se ha trabajado con HTML, JavaScript, CSS o AJAX, tecnologías ampliamente usadas en los sistemas web actuales.

Bases de datos: también se ha trabajado con diversos sistemas de bases de datos, ampliando los conocimientos que ya se poseían sobre SQL, y adquiriendo experiencia en administración de bases de datos. También se ha familiarizado con el uso de PL/SQL para bases de datos Oracle.

XML y XSLT: como lenguajes de tratamiento de datos y conversión de plantillas para creación de múltiples documentos.

11.3. Herramientas utilizadas

Durante el desarrollo de Genereitor se ha hecho uso de diferentes herramientas:

Entornos integrados de desarrollo: NetBeans 6.0, Oracle Jdeveloper 10.1.3.

Administración de bases de datos: Oracle SQLdeveloper 1.5, TOra.

Herramientas de construcción y despliegue: Apache Ant.

Sistemas de control de versiones: SVN.

Documentación: LaTeX.

Sistemas de bases de datos: Oracle, SQLServer, HSQLDB, MySQL, PostgreSQL.

Servidores de aplicaciones: Oracle Container for Java (OC4J), Apache Tomcat.

Navegadores: Firefox 2, Firefox 3.

11.4. Datos del proyecto

En la tabla 11.4 en la página siguiente se presentan unos cuantos datos numéricos relativos a las plantillas utilizadas por Genereitor, así como de los paquetes de datos que genera, que dan una visión aproximada de la cantidad de código que es utilizado para proporcionar dichos paquetes a la medida de cada aplicación.

Datos de las plantillas					
	Núm. plantillas	Líneas totales	Tamaño total	Promedio líneas/plantilla	Promedio tamaño/plantilla
Esqueleto	135 plantillas	≈ 24.400 líneas	1,1 MiB	≈ 180 líneas/plantilla	8,5 KiB/plantilla
Código	28 plantillas	≈ 5.700 líneas	384,2 KiB	≈ 200 líneas/plantilla	14 KiB/plantilla
Tablas Maestras	2 plantillas	257 líneas	16,5 KiB	≈ 130 líneas/plantilla	8,5 KiB/plantilla
TOTALES	165 plantillas	≈ 30.300 líneas	1,5 MiB	≈ 185 líneas/plantilla	9,3 KiB/plantilla

Datos de los paquetes generados					
	Núm. archivos	Líneas totales	Tamaño total	Promedio líneas/archivo	Promedio tamaño/archivo
Esqueleto[a]	116 archivos	≈ 11.400 líneas	714 KiB	≈ 100 líneas/archivo	6,2 KiB/archivo
Código[b]	18 archivos	≈ 3.500 líneas	260 KiB	≈ 195 líneas/archivo	14,5 KiB/archivo
Tablas Maestras[c]	4 archivos	≈ 890 líneas	85 KiB	≈ 222 líneas/archivo	21 KiB/archivo

Cuadro 11.1: Datos de las plantillas de Genereitor y los paquetes generados.

[a]Los datos del paquete de esqueleto no incluyen imágenes ni librerías.
[b]Los datos se han tomado analizando el paquete generado para una sóla tabla de 5 campos.
[c]Los datos se han tomado analizando el paquete generado para tres tablas relacionadas.

Capítulo 12

Trabajo futuro

12.1. Mantenimiento

Durante el proceso de diseño y desarrollo de Genereitor siempre se tuvo en cuenta que no iba a ser una herramienta «definitiva», sino que por el contrario estará siempre sometida a labores de mantenimiento, que corresponderán con los cambios en los requisitos y las exigencias de los clientes, y la futura adopción de nuevas tecnologías o actualización de versiones de las utilizadas actualmente, por lo que habrá que adecuar el funcionamiento de Genereitor a las nuevas circunstancias.

Es por esto que se ha previsto un sistema muy modular, de manera que cada componente realice funciones simples de forma eficaz y transparente, lo que facilita enormemente las tareas de mantenimiento.

12.1.1. Adición de nuevas características a Genereitor

12.1.1.1. Nuevos sistemas de bases de datos.

Como ya se ha visto, Genereitor soporta actualmente cinco sistemas de bases de datos (Oracle 9, PostgreSQL, MySQL, SQLServer y HSQLDB). Si en un futuro se desease dar soporte a un nuevo sistema de bases de datos, sería necesario implementar su capa de acceso a datos (recordemos que existe una fachada común a todos, pero una implementación para cada uno, puesto que son manejados de manera diferente) y los componentes que manejan su conexión. También se habrá de añadir dicho sistema de bases de datos al seleccionable de los formularios de la interfaz correspondientes a las vistas de conexión.

Al implementar Genereitor se ha tenido en cuenta la posibilidad de que en un futuro sea necesario añadir, por lo que se ha hecho especial hincapié en la modularidad de la gestión de los diferentes sistemas de bases de datos. Por ejemplo, hay componentes que se han implementado de forma redundante, siendo iguales para varios sistemas de bases de datos. Esto es debido a que a la hora de añadir soporte para un sistema de bases de datos nuevo, se podrá copiar la

implementación de uno ya existente y modificarlo según convenga, de manera que los componentes que manejan los sistemas existentes no tengan que ser modificados, y no se ponga en riesgo el perfecto funcionamiento de dichos sistemas durante la implementación del soporte para el nuevo.

12.1.1.2. Nuevos bloques funcionales

Genereitor cuenta con tres bloques funcionales, pero se contempla que en un futuro disponga de otros servicios relacionados con el desarrollo de nuevos proyectos.

Dada la modularidad que ofrece la plataforma J2EE, implementar y acoplar un nuevo bloque funcional sería una tarea trivial, únicamente siendo necesaria la adición de un enlace en el tag *contenedor* para que apareciera un botón en la barra superior que llevase hasta la nueva funcionalidad, y una vez implementada la capa de aplicación del nuevo módulo, la compilación y el despliegue se haría de forma automática.

El nuevo bloque funcional dispondría de las funcionalidades implementadas para la capa de acceso a datos, con lo que podría acceder de forma transparente a diversos sistemas de bases de datos, y utilizar los objetos Conexion, Tabla y Columna. También contaría con la colección de *custom tags* y estilos para implementar las vistas de la interfaz, de forma que la apariencia del nuevo bloque funcional fuera similar al resto de la aplicación.

12.1.2. Mantenimiento y modificación de los proyectos generados.

Todos los ficheros fuente se generan a partir de plantillas XSL, de forma que cualquier modificación en los requisitos de los proyectos a generar que implique una modificación en el código fuente, se aplicará directamente sobre las plantillas.

Las plantillas XSL se combinan con cadenas que contienen los datos, extraídos bien de la base de datos sobre la que se va a ejecutar la aplicación, bien de los parámetros introducidos por el programador en la interfaz de Genereitor. Estas cadenas están diseñadas con XML, teniendo los parámetros de cada nodo nombres intuitivos, precisamente para facilitar su adición a las plantillas.

También es importante destacar el hecho de que actualmente las cadenas XML contienen exceso de información respecto a la utilizada en las plantillas. La razón de este exceso es precisamente la potencial necesidad de usar esas información «extra» en un futuro, de forma que ya esté disponible y no sea necesario modificar las funciones que recolectan los datos, reduciendo este mantenimiento únicamente a la modificación de las plantillas, lo que supone una ventaja que contrarresta con creces la disminución del rendimiento de las combinaciones[1].

[1]Dada la velocidad de las combinaciones de las plantillas XSL, el exceso de datos en los nodos provoca una disminución de rendimiento tan pequeña que es irrelevante, sobre todo teniendo en cuenta que el rendimiento general de Genereitor no es un aspecto crucial, puesto que no es una herramienta de uso intensivo.

Así, cualquier modificación que sea necesaria en el código generado deberá ser llevada a cabo en las propias plantillas. El uso de XSL permite que estas modificaciones puedan ser llevadas a cabo de una forma intuitiva, ya que es un lenguaje sencillo y que diferencia bien los elementos que manejan el flujo de la transformación (las sentencias XSL) de los componentes de la plantilla (elementos en el lenguaje correspondiente al fichero que va a ser generado).

Las modificaciones en la estructura de archivos de los proyectos generados deberán ser llevadas a cabo en los propios servlets de Genereitor, ya sean en la fase de *generación de esqueleto*, *código* o *tablas maestras*. Dichas rutas se hallan agrupadas dentro de los servlets en *arrays* según las condiciones del proyecto a generar, separando los grupos de plantillas y ficheros generados que se han de combinar para cada opción seleccionada en la interfaz.

Es importante tener en cuenta que, de producirse una modificación en la estructura de los proyectos generados, los paquetes que devuelven los tres bloques funcionales han de ser modificados para conservar una estructura idéntica, ya que de lo contrario se producirán fallos en el árbol de directorio, generando archivos por duplicado, producto de la mezcla de la estructura nueva y la vieja.

12.1.3. Trabajo futuro

El desarrollo que se ha de llevar a cabo de manera más inmediata es la integración en los proyectos generados de los módulos CMS[2], XMI y el soporte para grupos y perfiles del módulo Usuarios. Todos estos módulos han sido desarrollados en la Empresa y son utilizados en muchas de las aplicaciones que se desarrollan.

Posteriormente, y según las necesidades con respecto a los requisitos de nuevos proyectos que se contraten, se ampliará el soporte para otros sistemas de bases de datos tal como se ha comentado anteriormente, o se integrarán nuevos módulos que se desarrollen en la empresa para uso común.

[2]Actualmente el desarrollo de soporte para CMS se halla muy avanzado

Parte VI

Apéndices

Apéndice A

Documento de análisis y diseño

A continuación se presenta la transcripción del documento de **Análisis y Diseño** elaborado previamente al comienzo del desarrollo de la aplicación.

> **Nota:** La elaboración del documento de *análisis y diseño* es previa al comienzo del desarrollo del proyecto, por lo que existen bastantes diferencias entre lo que en un principio se planeó —y que aparece en este documento— y el resultado final.

A.1. Descripción y objetivos

El objetivo de este proceso es la obtención de una especificación detallada del nuevo **Generador de código J2EE**, de forma que éste satisfaga todas y cada una de las necesidades funcionales de los usuarios de la aplicación.

En el apartado de este documento *Situación Actual* se realiza un estudio sobre cómo trata el sistema actual el proceso de generación de código para de esta manera poder detectar problemas y necesidades que el nuevo sistema de información tratará de solventar tanto a nivel de funcionalidad como de agilidad en el manejo de la herramienta, así como añadir nuevas características que permitan incrementar su eficiencia y evolucionar ciertos apartados del sistema actual para adaptarlos a las nuevas tendencias tecnológicas.

La definición del sistema junto con la especificación de requisitos realizada en este proceso permitirá describir con precisión el nuevo sistema de información, entrando en detalle en las tareas propias del Análisis Funcional:

- Modelo de procesos del sistema.
- Interfaces de usuario.

La participación activa de los usuarios durante las diferentes sesiones de trabajo para la completa definición del sistema habrá resultado decisiva e imprescindible para la actual fase de análisis del nuevo sistema, ya que dicha participación constituye una garantía de que los requisitos, datos, procesos e interfaces identificados han sido comprendidos e incorporados al sistema.

El objetivo del proceso de *Diseño del Sistema de Información* es la definición de la arquitectura del sistema y del entorno tecnológico que le va a dar soporte, junto con la especificación detallada de los componentes del sistema de información.

A partir de dicha información, se generan todas las especificaciones de construcción relativas al propio sistema, así como la descripción técnica del plan de pruebas, la definición de los requisitos de implantación y el diseño de los procedimientos de migración y carga inicial.

A.2. Resumen ejecutivo

El objetivo de este proyecto es ofrecer una herramienta que permita agilizar la creación de nuevos proyectos consistentes en aplicaciones web contra bases de datos de cualquier tipo.

Se trata de una plataforma web que genera código fuente J2EE de una aplicación web a todos los niveles, así como los elementos para los diferentes niveles del esquema:

A.2.1. *Data Access Layer* (Capa de Acceso a Datos)

Genera las clases de acceso a datos con las operativas *CRUD* más comunes: inserción de registros, actualización, consulta (por clave primaria o por cualquier otro campo) y borrado (unitario y masivo).

A.2.2. *Business Logic* (Lógica de Negocio)

La plataforma ha de dar opción de implementar la BL mediante *Enterprise Java Beans* (EJBs) o mediante clases básicas Java, dependiendo si el servidor de aplicaciones en el que irá alojado el proyecto soporta EJBs o no.

A.2.3. *Model-View-Controller* (Modelo-Vista-Controlador)

La interfaz de usuario será implementada siguiendo el patrón de modelo-vista-controlador. Para ello se utilizarán clases previamente implementadas en el `comex-common`[1]. Se generarán JSPs de listado y de alta/modificación, y *Actions* (*Pseudo-Servlets*).

[1] Comex dispone de un conjunto de clases y utilidades desarrolladas por la propia Empresa, que se agrupan en el paquete `comex-common`.

Para la realización de los JSPs se utilizará la colección de etiquetas (*tags*) implementados en el `comex-common`. Asímismo, estos JSPs han de cumplir obligatoriamente con un nivel AA de accesibilidad.

El proyecto es una plataforma capaz de crear la base de una aplicación web apoyada en una base de datos, capaz de realizar los accesos CRUD típicos a ésta. También crea la estructura de ficheros básica en la que se asientan todos los proyectos realizados por el departamento Java de Comex Grupo Ibérica, así como los archivos de instalación y configuración de la aplicación que se va a crear.

El objetivo de este proyecto es facilitar en la medida de lo posible el comienzo del desarrollo de nuevas aplicaciones requeridas por los clientes de Comex Grupo Ibérica, proporcionando una base estándar, pero personalizada conforme a los requerimientos del sistema solicitado por el cliente. De este modo se evita el tener que comenzar cada nuevo proyecto desde cero, creando la estructura de ficheros y proporcionando una parte importante del código fuente, ajustando ya a los parámetros de la nueva aplicación, de modo que el programador no tenga que recurrir a un proyecto ya existente y modificar su código para ajustarlo a la configuración y necesidades del nuevo programa.

Esta herramienta proporciona un ahorro de tiempo considerable en las primeras etapas de desarrollo, permitiendo al programador ocupar su tiempo en tareas más específicas de cada proyecto, y evitándole las tareas mecánicas de creación de las bases de cada aplicación que se desarrolla, ya que el hecho de que exista una base común ya creada por la organización permite automatizar esta tarea.

El funcionamiento se basa en la consulta de la base de datos sobre la que se apoyará la aplicación. Esta base de datos puede ser proporcionada por el cliente, rescatada de un sistema de datos anterior del propio cliente o bien desarrollada por Comex a partir de los requisitos del proyecto solicitado. En cualquier caso, la elaboración de esta base de datos es un paso previo al uso del Generador, ya que éste proporciona un sistema de consulta y edición de los contenidos de las tablas donde se almacenan los datos, pero no es capaz de editar las propias tablas o añadir nuevas.

Es una herramienta de ayuda a la implementación de la aplicación requerida por el cliente, presentando una interfaz web que presenta las diferentes relaciones de la base de datos, así como las diferentes opciones disponibles para generar la aplicación, escogiendo el programador las que más se ajusten al proyecto solicitado por el cliente.

Una vez elegidas las opciones, el Generador devuelve un paquete comprimido, conteniendo la estructura de archivos de la aplicación y los métodos de acceso a la base de datos ya implementados, todo listo para descomprimir en el servidor de pruebas y comenzar con el desarrollo de los apartados de la aplicación que no sea posible generar de forma automática, como podría ser la interfaz web que se presentará a los usuarios finales de la aplicación, etc.

A.3. Sistema Actual

Se dispone actualmente de un generador de código, que ofrece soporte para aplicaciones apoyadas en bases de datos Oracle, MySQL, PostgreSQL, HSQLDB y SQLServer. Al conectar a la base de datos, ofrece un *combo* con la lista de tablas presentes en la base de datos.

Figura A.1: Página de conexión del generador actual

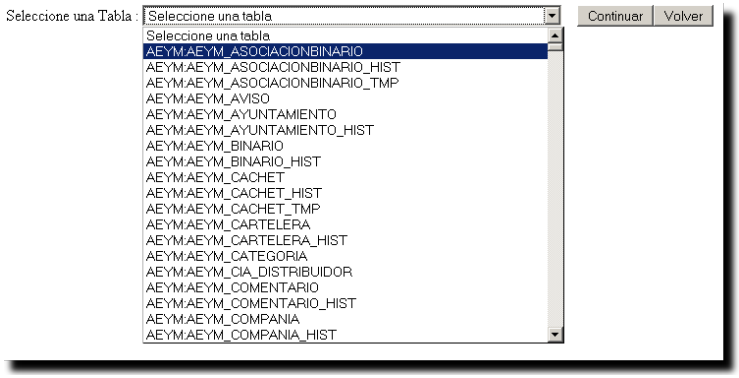

Figura A.2: Página de selección de tabla del generador actual

El primer problema con el que nos encontramos es que cuando se produce un error de conexión con la base de datos, el programa no devuelve un mensaje de error, sino que vuelca a la página la excepción provocada.

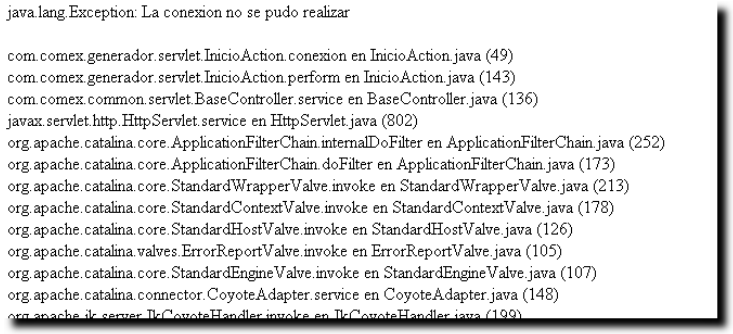

Figura A.3: Error de conexión del generador actual

Una vez conectado a la base de datos correctamente y seleccionado una tabla de la misma, se nos presenta una pantalla con todos los campos que la forman, señalando las claves primarias. También ofrece información sobre el tipo de dato que se almacena en cada campo, su longitud y decimales en caso de ser valores numéricos.

A continuación se ofrecen los distintos parámetros que podemos configurar para generar el código apropiado a las necesidades del proyecto.

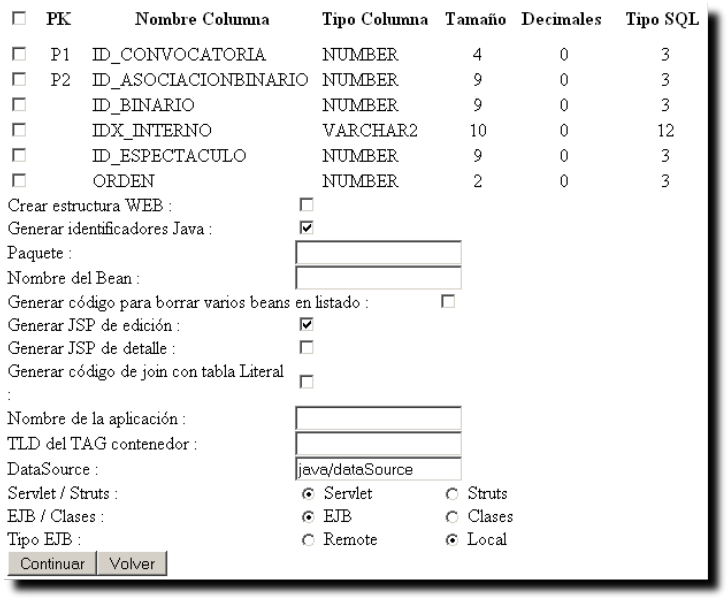

Figura A.4: Página de selección de campos y parámetros del Generador actual

Los problemas más importantes en el Generador actual son los siguientes:

- No se tiene en cuenta ningún tipo de relación entre tablas, lo cual se traduce en que todas las relaciones (claves ajenas) existentes en la base de datos han de ser escritas a mano por el programador para que la nueva aplicación las tenga en cuenta y se mantenga la consistencia en los datos.
- No se ofrece en el generador la posibilidad de añadir restricciones al programa que no tenga la base de datos (obligar en el programa a introducir valores en campos que la base de datos acepta como nulos, incrementa las restricciones en longitudes de campos, añade comprobaciones en campos que contengan NIF o fechas).
- Se ofrecen opciones que ya no son útiles o ya no interesa tenerlas en cuenta, como la posibilidad de usar *Struts* en lugar de servlets, que no se incluirán en el nuevo Generador.
- La opción de crear la estructura del proyecto se ofrece en la última etapa del proceso, haciendo necesario conectar a una base de datos y seleccionar

una tabla, pese a que a la hora de crear el esqueleto no se utiliza para nada esta conexión.

A.4. Identificación y definición de requisitos

En este apartado se lleva a cabo la definición, análisis y validación de los requisitos a partir de la información facilitada por los usuarios participantes. Se obtiene un catálogo detallado de los requisitos, a partir del cual se puede comprobar que los productos generados en las actividades de modelización se ajustan a los requisitos de usuario.

A.4.1. Ámbito y alcance del proyecto

A.4.1.1. Definición del proyecto

El proyecto será una plataforma web capaz de generar código J2EE automáticamente ajustándose a los requisitos del proyecto solicitado por el cliente y cuyo desarrollo se va a comenzar.

Para su funcionamiento será necesaria la creación previa de una base de datos contra la que trabajará la nueva aplicación, dado que esta plataforma no ha de ser capaz de modificar las bases de datos que tiene en cuenta al generar el código, y por tanto tampoco de crearlas (las aplicaciones generadas con esta herramienta sí han de poder modificar su base de datos).

El Generador trabajará sobre el servidor de aplicaciones *Apache Tomcat*, y será capaz de manejar los tipos de bases de datos más comunes (Oracle, PostgreSQL, MySQL, HSQLDB, SQLServer), contemplando la posibilidad de añadir soporte para otros sistemas de bases de datos con la simple inclusión de un plugin. También podrá crear código para aplicaciones que serán soportadas tanto por Tomcat como por otros servidores de aplicaciones (OC4J, etc), según lo indiquen los requisitos del proyecto.

Los proyectos generados con esta aplicación seguirán la arquitectura en tres capas de J2EE (figura A.5 en la página siguiente), compuesta por una capa de acceso a datos, la lógica de negocio y el patrón modelo-vista-controlador.

La capa de acceso a datos proporciona un acceso simplificado a la información contenida en la base de datos contra la que trabajará la aplicación que se está creando. Esto permite que los demás componentes del sistema accedan a la base de datos sin necesidad de conocer el modo de almacenamiento de ésta, de modo que se puedan tratar los objetos como tales, en lugar de como una tupla con valores contenida en una tabla de la base de datos. Esto proporciona un nivel mayor de abstracción.

Sobre esta capa de acceso a datos está la lógica de negocio, consistente en una serie de componentes que interactúan con la capa de acceso a datos, y una fachada que simplifica el uso de estos componentes y contra la que trabajan las actions de la tercera capa. La lógica de negocio se podrá implementar mediante EJBs o clases básicas de Java, según dicten los requisitos del proyecto que solicita

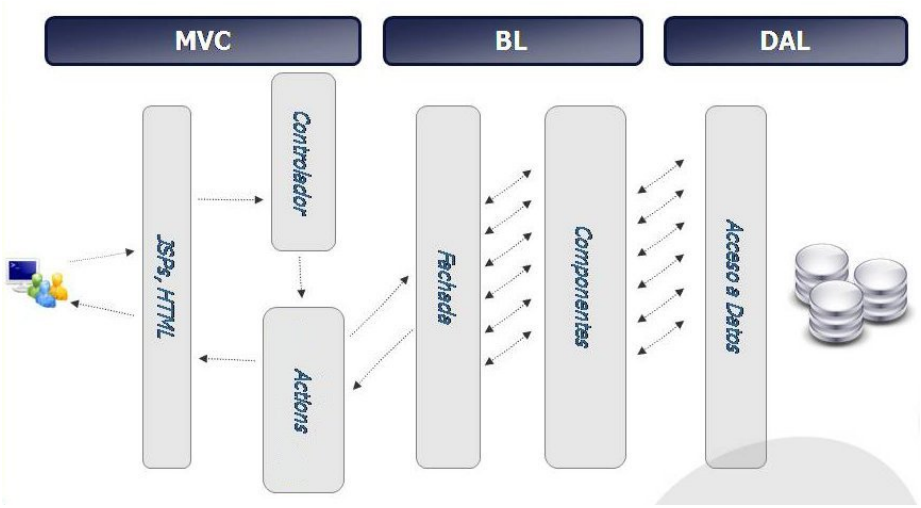

Figura A.5: Esquema de la arquitectura J2EE

el cliente, y si el servidor sobre el que funcionará la aplicación soporta EJBs o no.

La tercera capa sigue el patrón modelo-vista-controlador, de forma que el usuario del programa interactuará con una vista, comúnmente compuesta por páginas JSP. Estas JSPs harán peticiones al servidor, quien tiene implementado un controlador que se mantiene a la espera de instrucciones. En el momento en que recibe una petición, invoca las actions necesarias, que se comunicarán con la capas inferiores y devolverán al usuario el resultado de la operación requerida. La interfaz de acceso público generada por el programa cumplirá obligatoriamente con un nivel de accesibilidad AA[2] según el W3C.

Al solicitar el usuario la creación del esqueleto de un nuevo proyecto, aparte de la estructura de ficheros de la aplicación, también serán generados los ficheros de configuración XML y los scripts de instalación que luego serán usados por `ant` para instalar la aplicación en el servidor.

A.4.1.2. Identificación de participantes (Actores)

El usuario final de esta herramienta será el desarrollador del proyecto requerido por el cliente, puesto que el Generador es una ayuda al desarrollo de las aplicaciones solicitadas a Comex Grupo Ibérica.

Aunque la estructura general del proyecto pueda ser creada por un Jefe de Proyecto o un Analista y los diferentes módulos sean generados por los Programadores, en la aplicación sólo existe un perfil de usuario.

[2]Aunque las interfaces generadas por el programa cumplan con el nivel AA de accesibilidad, la propia interfaz del Generador no lo cumplirá, puesto que para facilitar tareas a la hora de usarlo se utilizará JavaScript.

A.5. Catálogo de requisitos del sistema

El nuevo sistema a desarrollar tiene como objetivo cumplir una serie de requisitos funcionales y no funcionales que se resumen a continuación:

A.5.1. Requisitos funcionales

1. El sistema generará el esqueleto inicial de la aplicación el cual podrá ser para una aplicación pensada para correr sobre un servidor que posea contenedor de EJBs o no.
2. El esqueleto inicial de la aplicación contendrá los descriptores de ficheros XML necesarios para el empaquetado mediante la herramienta `ant`.
3. En el esqueleto inicial vendrán implementados los componentes genéricos de la aplicación:
   - Fachada.
   - Acción base.
   - Clases con constantes.
   - Ficheros de configuración (acciones, textos).
   - Ficheros de etiquetas.
   - Descriptores de ficheros (para contenedores de EJBs si es el caso, para el servidor web y *taglibs*).
   - Librerías necesarias.
4. En el esqueleto inicial se ha de incluir también las clases, librerías y configuraciones necesarias para el uso de las herramientas genéricas para la gestión de parámetros y tablas maestras implementadas por Comex.
5. El sistema generará el código necesario para todas las capas tomando como base una de las tablas de la base de datos.
6. El código generado dependerá de si el servidor sobre el que se va a ejecutar soporta EJBs o no.
7. Si el servidor soporta EJBs, se dará la opción de generar código para EJBs locales o remotos.
8. El sistema tendrá en cuenta las relaciones entre las tablas de la base de datos (*foreign keys*), y generará el código para acceder a las tablas referenciadas por éstas automáticamente.
9. El sistema ha de ser capaz de generar aplicaciones para diferentes sistemas de bases de datos (Oracle, PostgreSQL, MySQL, HSQLDB, SQLServer).
10. Existirá la posibilidad de añadir soporte para otros sistemas de bases de datos mediante la inclusión de *plugins* al sistema. Se proporcionará documentación para ello.

11. Generará aplicaciones pensadas para ser soportadas por diferentes servidores de aplicaciones (Apache Tomcat, OC4J).

12. Creará las bases de la interfaz web genérica con sus distintos apartados (cabeceras y pies comunes a todas las páginas, menú).

13. Proporcionará una serie de funciones JavaScript genéricas para la interfaz web (calendario y comprobación de fechas, NIF/CIF, implementación del algoritmo md5...).

A.5.2. Requisitos no funcionales

1. Toda interfaz de acceso público generada por el programa cumplirá con un nivel de accesibilidad AA[3] según el W3C.

2. El aplicativo web será compatible con los navegadores web estándar.

3. La arquitectura del software será la de un *Sistema Centralizado con Arquitectura de Tres Capas*. Dicha arquitectura seguirá los estándares J2EE.

4. El servidor web que soportará la aplicación será *Apache Tomcat*.

5. La compilación e instalación de las aplicaciones generadas se llevará a cabo mediante `ant`, que proporciona independencia de la plataforma sobre la que se instala, al contrario que `makefile`.

A.6. Modelo de procesos

Para la realización de este apartado se habrán analizado las necesidades de los usuarios para establecer un conjunto de procesos que conformen el sistema de información. Se realizará un enfoque descendente (*top-down*), en varios niveles de abstracción y/o detalle, donde cada nivel proporciona una visión más detallada del proceso definido en el nivel inmediatamente superior.

En la elaboración de este modelo de procesos se llegará a un nivel de descomposición en el que los procesos obtenidos sean claros y sencillos, es decir, buscando un punto de equilibrio en el que dichos procesos tengan significado por sí mismos dentro del sistema global y a su vez la máxima independencia y simplicidad.

La totalidad de procesos indicados en este apartado serán los que deberá dar soporte el nuevo sistema.

A.6.1. Generación del esqueleto

Al iniciar el Generador, se ofrecerá al usuario la opción de crear el sistema de ficheros básico para una nueva aplicación, dado que para esta tarea no se requiere conexión a base de datos.

[3]Únicamente es requisito que sean accesibles las interfaces de acceso público, es decir, la parte de la interfaz destinada a administración no ha de cumplir este requisito.

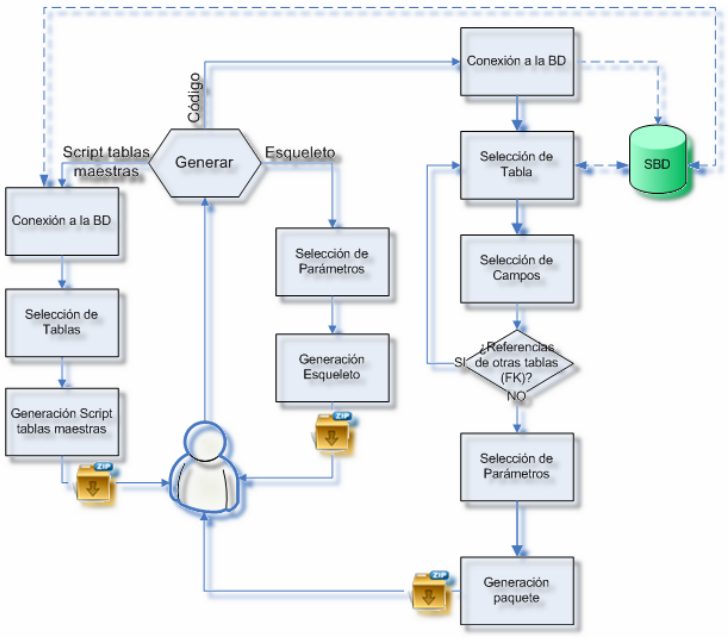

Figura A.6: Esquema de los procesos del Generador

Una vez seleccionadas las opciones correspondientes, se devuelve un fichero comprimido que contiene la estructura «vacía» del nuevo proyecto, así como los ficheros XML de configuración y los scripts de instalación que luego serán usados por `ant` para instalar la aplicación en el servidor que corresponda.

Aunque no se necesite conectar a la base de datos para generar el esqueleto, la aplicación pedirá una serie de datos necesarios para crear la estructura:

- Tipo de base de datos (*combo* de los tipos de BD disponibles en el Generador).
- Tipo de servidor de aplicaciones (*combo* de los tipos de servidores disponibles en el Generador).
- Nombre de la aplicación a crear.
- Nombre del paquete base.
- Datos de conexión a la base de datos (host, puerto, SID, usuario, contraseña). Opción de comprobar si estos datos son correctos mediante un botón que lance la conexión contra la base de datos y devuelva un mensaje según los datos sean correctos o no.
- Prefijos de los nombres de tabla (terminados en «_»).
- Prefijos de los parámetros (terminados en «_»).

- Si el esqueleto va a ser generado con parámetros o con tablas maestras (*radio*).
- Ruta de los parámetros/tablas maestras.
- Lista de librerías a incluir.

El esquema de procesos de la fase de generación del esqueleto se presenta en la figura A.7.

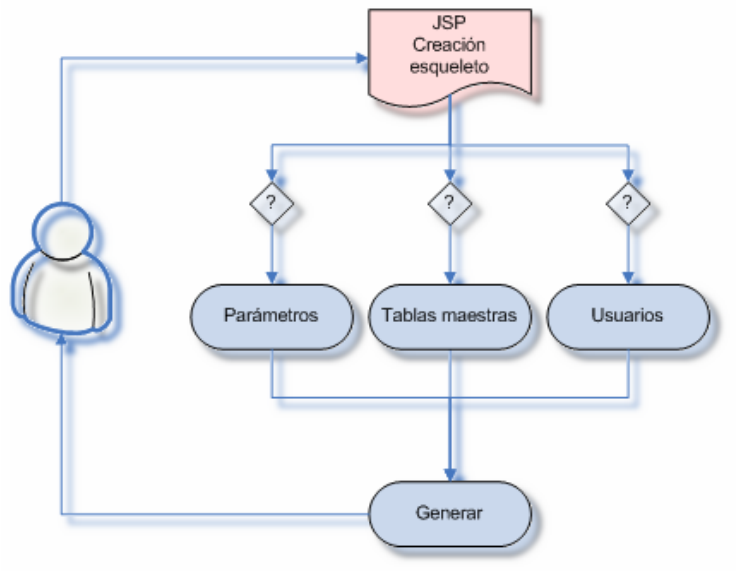

Figura A.7: Esquema de procesos de la generación del esqueleto

A.6.2. Generación de script de tablas maestras

Esta funcionalidad del generador permite crear el script de tablas maestras para un proyecto mediante la conexión a la base de datos sobre la que se apoyará, permitiendo al usuario seleccionar las tablas que se habrán de tener en cuenta para la generación de dicho script.

En primer lugar se presenta una pantalla de conexión, donde una vez introducidos los datos de host, puerto, SID, usuario y contraseña, el programa se conecta a la base de datos y devuelve una lista de todas las tablas que la conforman (figura A.8 en la página siguiente). De dicha lista el usuario deberá elegir las que convengan para generar el script. Una vez seleccionadas las tablas apropiadas, el Generador ofrecerá al usuario una lista de todos los campos y opciones.

Una vez seleccionadas las tablas que se utilizarán, se presenta una pantalla con todos los campos que conforman las tablas seleccionadas, así como una serie

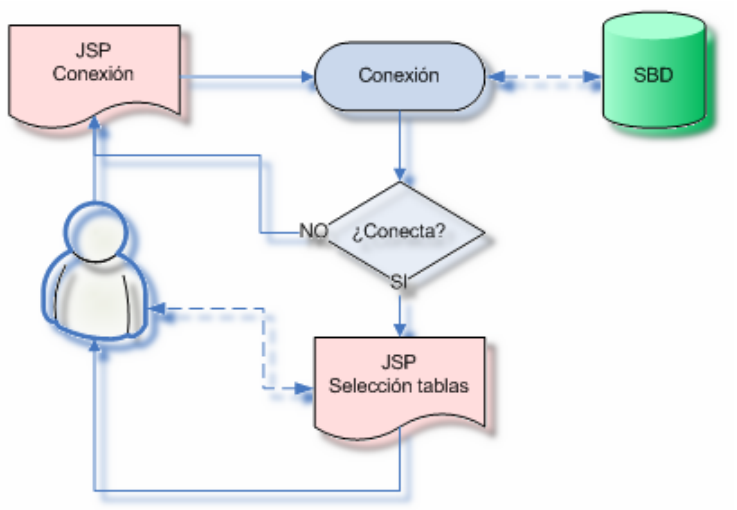

Figura A.8: Esquema de procesos de la selección de tablas para la generación del script de tablas maestras

de opciones tanto de cada tabla como de los campos, de forma que el usuario pueda elegir las que más convengan según los requisitos del proyecto a realizar.

Para cada tabla se informará de su clase y se pedirá una descripción (*literal*) y la selección del campo por defecto. Para cada campo, aparte de las opciones de tabla, se podrá aplicar la restricción de campo obligatorio y se presentarán las posibilidades de tenerlos en cuenta para listados, buscadores, detalles y el tratamiento de claves ajenas. Una vez escogidas todas las opciones necesarias, se genera el script.

El esquema de procesos de este último paso se presenta en la figura A.9 en la página siguiente.

A.6.3. Generación de código

A.6.3.1. Conexión a la base de datos

El primer paso para la generación de un nuevo módulo de un proyecto es la conexión a la base de datos sobre la que se ejecutará. Para esto, el usuario seleccionará el tipo de base de datos que la aplicación va a usar (y que debe estar implementada previamente al uso del Generador). Tras esto, rellenará los datos de conexión (Host, Puerto, SID, Usuario y Contraseña) y pulsará el botón «Conectar».

Al pulsar «Conectar» se comprobará que los datos de conexión introducidos sean correctos, de otro modo devolverá un error con una breve explicación y volverá a pedir los datos de conexión.

También se comprobará que el host contra el que lanzamos la petición esté activo, y que la base de datos que hemos introducido exista en el servidor, así como

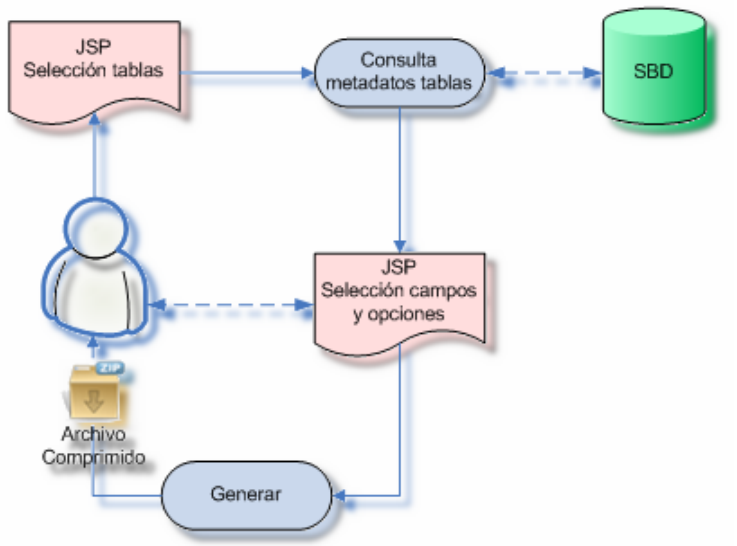

Figura A.9: Esquema de procesos de la selección de campos y opciones para la generación del script de tablas maestras

que el usuario y la contraseña proporcionados para el acceso a la misma sean correctos. Si algo de esto no se cumple, nuevamente se mostrará un error y retornará a la página de conexión.

A.6.3.2. Selección de la tabla principal

Tras conectar a la base de datos, el sistema realiza una consulta y se presenta al usuario un listado de todas las tablas contenidas, seleccionando éste la que le interese. Al seleccionarla, se realizará otra consulta a la base de datos, obteniendo el nombre de los campos que conforman esa tabla.

A.6.3.3. Selección de tablas relacionadas y campos

Una vez seleccionada la tabla principal, se vuelve a consultar la base de datos, mostrando al usuario un listado de las tablas que hacen relación a ésta, y ofreciendo la posibilidad de seleccionar o no cada una de ellas, de forma que sean tenidas o no en cuenta a la hora de generar el código.

Asimismo para cada tabla (principal y relacionadas) existe la posibilidad de consultar y seleccionar sus campos. Al seleccionar una de las tablas, se presentará al usuario el listado de campos que la forman, de manera que éste pueda elegir los que interesan. También se ofrecerá información acerca de tipos de datos almacenados, nombres de variables y si son clave primaria de la tabla.

Las consultas a base de datos se realizarán tal como indica el esquema de la figura A.10 en la página siguiente. Se consultarán las claves ajenas de otras

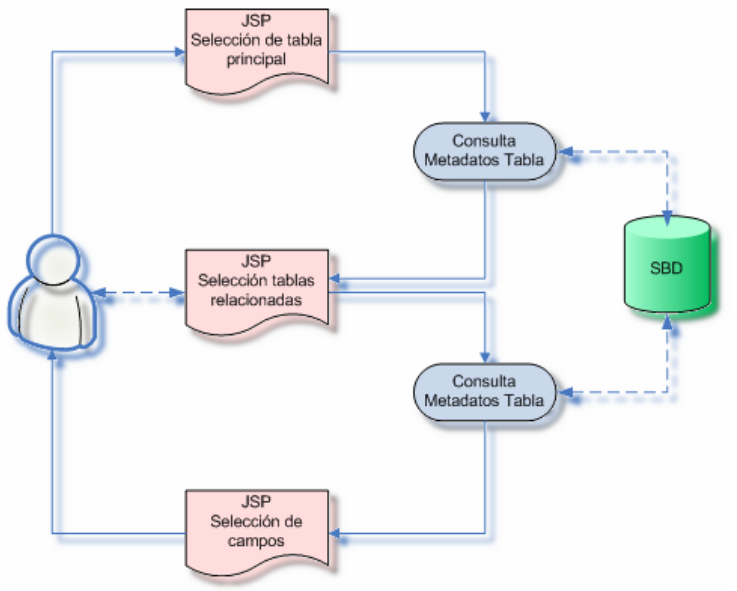

Figura A.10: Esquema de procesos de la selección de la tabla principal

clases que hagan referencia a la tabla a partir de la que se va a trabajar, y se consultará al usuario si quiere que sea tenida en cuenta. Sólo se tendrán en cuenta las referencias a primer nivel, es decir, se descartarán los atributos de las tablas que sean clave ajena de otra clase que haga referencia a la tabla sobre la que se está trabajando, de la forma que indica el diagrama de la figura A.11:

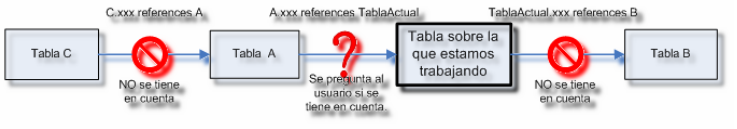

Figura A.11: Esquema de tratamiento de tablas relacionadas

Este proceso se repetirá tantas veces como clases hagan referencia a la clase seleccionada.

A.6.3.4. Selección de campos

Tras este paso, se ofrece al usuario una lista de todos los campos de las tablas que se han seleccionado con anterioridad, de manera que éste pueda revisar que todo sea correcto. En esta lista también se ofrecerán una serie de opciones de manera que se puedan aplicar algunas restricciones en la entrada de datos de la aplicación a generar que la base de datos no tenga.

La lista mostrará para cada tabla:

- Seleccionado: un *checkbox* que permite tener en cuenta o no los campos de cada tabla, tanto de la principal como de las relacionadas que se han seleccionado.
- ID del campo.
- Campo obligatorio: un *checkbox* nos indicará si en la base de datos ese campo puede tomar valores nulos o no. Si el campo es obligatorio (*mandatory*) en la base de datos, el *checkbox* aparecerá seleccionado e inactivo. Si no lo es, podremos seleccionarlo para que en nuestra aplicación sí lo sea.
- Clase: indica el tipo de dato que contiene cada campo.
- Clave: indica si el campo es clave primaria o ajena de alguna tabla.
- MaxLength: indica la longitud máxima que puede tomar un campo de tipo cadena de caracteres. Ofrece la posibilidad de aumentar en el programa generado la restricción que imponga la base de datos, o de establecerla si no existe.
- Descripción: permite introducir el nombre (*literal*) de cada campo.

A.6.3.5. Selección de parámetros

En este último paso se seleccionarán los parámetros oportunos para las necesidades de la aplicación que se va a generar. Las opciones son las siguientes:

- Nombre del paquete.
- Lista de nombres de *beans*.
- Generar código para borrar varios benas en listado (*check*).
- Generar JSP de edición (*check*).
- Generar JSP de detalle (*check*).
- Nombre de la aplicación.
- Generar con EJB/Clases Java (*radio*).
- Generar EJB locales/remotos (*radio*, sólo activo si en el anterior se seleccionó EJB).

A la hora de trabajar con las tablas relacionadas con la primaria, el único campo que variará respecto de ésta será el nombre del *bean* que tenga asociado. Es por esto que se presenta una lista con todos los nombres de las tablas seleccionadas, de forma que se puedan rellenar los diferentes nombres de los *beans*. Las demás opciones será idénticas para todas las tablas, así que sólo se tendrán que rellenar una vez.

Según las capacidades del servidor donde irá alojada la aplicación que estamos creando y los requisitos del proyecto, el generador ofrece la posiblidad de

generar código para usar EJBs o clases Java. Al seleccionar EJBs, aparecerá una nueva opción que dará a elegir entre EJBs locales o remotos. Si se seleccionan clases Java, esta opción no estará visible, ya que es indiferente.

Cuando ya hayamos completado todas las opciones para todas las tablas relacionadas que nos interesan, el Generador nos devolverá un paquete comprimido que contendrá el código fuente solicitado, tal como indica la figura A.12.

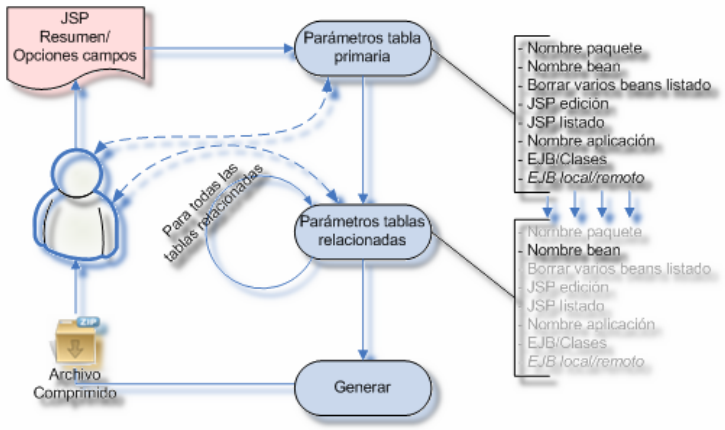

Figura A.12: Esquema de procesos de la selección de parámetros

A.7. Definición de la arquitectura del sistema

La estructura cliente-servidor posibilita que la aplicación sólo tenga que estar instalada en una máquina, que esperará a que los clientes realicen peticiones para ejecutar las operaciones requeridas y devolver los resultados. El servidor sobre el que está alojada la aplicación es *Apache Tomcat*, pese a que dentro de la funcionalidad del programa es capaz de generar código fuente para proyectos no basados en este servidor de aplicaciones.

La interfaz presentada al cliente es una plataforma web, de forma que sea accesible desde cualquier punto de la Intranet de la Empresa, o incluso a través de Internet, con un simple navegador, evitando la instalación de programas cliente específicos para cada usuario, y la actualización de estos programas cliente en posibles actualizaciones de la aplicación.

Las ventajas de este diseño frente a una estructura monolítica (un ejecutable autónomo, y una copia de éste instalada en cada cliente) son evidentes, haciendo que la disponibilidad del sistema sea instantánea en el momento que sea requerida, no teniendo que estar instalado en el cliente cuando no sea utilizado, y al ser presentado a través de una interfaz web, garantizamos el funcionamiento en cualquier sistema que pueda utilizar el cliente, puesto que las operaciones se ejecutarán sólo en el servidor.

Un inconveniente de esta arquitectura puede ser el riesgo de sobrecarga de la red o del servidor cuando muchos clientes accedan a él de forma simultánea,

pero dado que el uso de la aplicación no será intensivo, este aspecto puede ser descartado.

La arquitectura del proyecto, correspondiente con la arquitectura J2EE, está formada por tres capas (figura A.13 en la página siguiente):

El primer nivel, la capa de acceso a datos (DAL, Data Access Layer), es capaz de almacenar bases de datos de varios tipos (Oracle, MySQL, PostgreSQL, HSQLDB, SQLServer), según las propiedades o los requisitos del proyecto a realizar. Puesto que la elaboración de la base de datos es anterior al uso del Generador, éste ha de ser versátil en este sentido, ya que se ha de ajustar a las características requeridas por el proyecto solicitado por el cliente. Esta capa no es la encargada de manejar las conexiones a la base de datos, sino que siempre la recibirá como parámetro, siendo la lógica de negocio quien gestione la comunicación con ésta.

El segundo nivel, la lógica de negocio (BL, Business Logic), implementa los modos de acceso a la capa de datos, de forma que el programa sea capaz de trabajar con los distintos tipos de base de datos mencionados anteriormente. Asimismo, proporciona una fachada para la tercera capa, con arquitectura modelo-vista-controlador, de manera que el modelo sea capaz de comunicarse con los datos independientemente de la forma en que estén almacenados.

El tercer nivel, la parte web de la aplicación, sigue el patrón modelo-vista-controlador (MVC, Model-View-Controller). Esta separación en tres niveles permite independencia entre los datos presentados o requeridos por la interfaz de usuario (vista) y los que son enviados al servidor para su tratamiento (modelo), dado que es el controlador quien procesa y responde a los eventos invocados por la interfaz de usuario, y en su caso ordena lecturas o cambios en el modelo, que a su vez se comunicará con la *Business Logic* para consultar o actualizar los datos almacenados físicamente en la base de datos. Para el desarrollo de este nivel, se utilizan clases implementadas en el `comex-common`. La vista estará implementada mediante JSP.

A.7.1. Gestor de base de datos

El Generador de código tomará como base inicial los datos contenidos en una base de datos, que variará según las necesidades del proyecto a realizar. Dado que los diferentes proyectos se apoyarán sobre bases de datos de distintos tipos, el Gestor de bases de datos tendrá que cumplir con los siguientes requisitos:

- Compatible con SQL.
- Capaz de hacer consultas, pero ha de impedir la posibilidad de insertar, actualizar o borrar campos.

A.7.2. Servidor de aplicaciones

El sistema se implementará sobre un servidor de aplicaciones (*Apache Tomcat)*, que proporcionará a la aplicación web los siguientes recursos fundamentales:

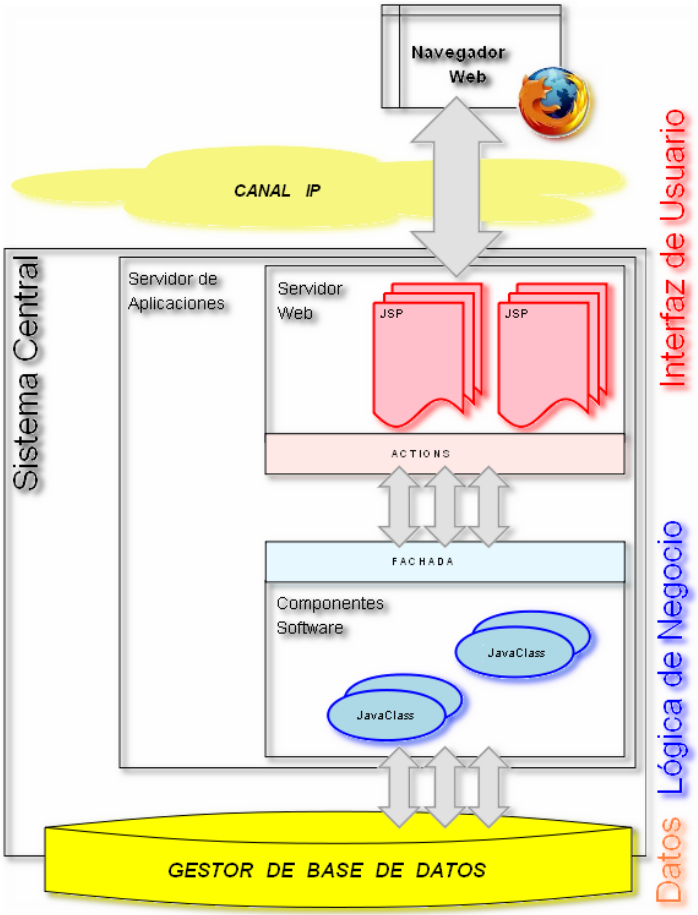

Figura A.13: Esquema de la arquitectura de software

Servidor web, capaz de construir de forma dinámica la interfaz del sistema, basado en páginas web (JSPs). Será capaz de soportar concurrencia y controlar distintas sesiones de trabajo simultáneas.

Contenedor, capaz de ejecutar los diferentes componentes de software que permitirán desarrollar la lógica de los componentes base y los módulos funcionales de la plataforma, así como encapsular la comunicación del sistema con el gestor de base de datos. Al igual que el servidor web, es capaz de soportar concurrencia y controlar distintas sesiones de trabajo simultáneas.

A.8. Modelo de datos

Dentro de este apartado se identifican todas las necesidades de información de cada uno de los procesos que conforman el sistema de información, con el fin

de obtener un modelo de datos que contemple todas las entidades, relaciones y atributos necesarios para dar respuesta a dichas necesidades.

Dado que este proyecto no se apoya en ninguna base de datos propia, no existe un modelo de datos concreto que podamos describir de antemano.

El Generador es capaz de trabajar con bases de datos a dos niveles:

A.8.1. Consulta

Para generar el código de la aplicación, el generador se ejecuta contra la base de datos propia de la aplicación a crear, cuya implementación se habrá realizado de antemano. Esta consulta no va dirigida a los datos que pudieran estar contenidos en esa base de datos, sino a los **metadatos** propios de su estructura. Interesa *cómo* están almacenados los datos.

Por ello, el sistema ha de ser capaz de realizar estas consultas en cualquier sistema de bases de datos contra el que sea ejecutado, o en su defecto prever una ampliación para sistemas de bases de datos que no estén incluidos en la implementación base, de manera sencilla y documentada, mediante *plugins* que serían desarrollados conforme a las nuevas necesidades.

A.8.2. Generación

Una vez consultada la estructura de la base de datos, el Generador será capaz de crear órdenes que se efectúen contra la base de datos, ya no sólo de consulta, sino de modificación, creación y borrado de registros, esta vez ya a nivel de contenidos (datos) y no de estructura (metadatos). Esto es debido a que las aplicaciones generadas han de ser capaces de modificar los datos contenidos en las bases de datos contra las que trabajan.

A.9. Definición del interfaz de usuario (prototipos)

La interfaz de usuario está estructurada en tres «ramas», correspondientes a las tres funciones principales del Generador:

- Generación del **esqueleto** de un proyecto.
- Generación del **script de tablas maestras** de un proyecto.
- Generación de **código** de las diferentes partes de un proyecto.

Estas tres opciones se presentan en la página inicial del Generador, mediante tres botones que enlazan a cada una de las funciones. Existe una cuarta opción (*Ajustes*), que permitirá establecer los parámetros de conexión por defecto para las distintas bases de datos sobre las que la Empresa esté trabajando en cada momento, de forma que no se tengan que introducir cada vez que se requiera la

ayuda del Generador. A continuación se detallan los diferentes componentes de cada pantalla de la interfaz.

A.9.1. Generación del esqueleto

Esta parte consta sólo de una pantalla, que pedirá los siguientes campos:

Tipo de base de datos: será un *combo* que mostrará los sistemas de bases de datos para los que puede trabajar el Generador (Oracle, MySQL, PostgreSQL, HSQLDB, SQLServer y cualquiera que se añada en un futuro), de los que habrá que seleccionar el que soportará la nueva aplicación.

Tipo de servidor de aplicaciones: será un *combo* que mostrará los servidores de aplicaciones para los que puede trabajar el Generador (OC4J, Tomcat y cualquiera que se añada en un futuro), de los que habrá que seleccionar el que soportará la nueva aplicación.

Nombre de la aplicación: será un *textbox* en donde habrá que introducir el nombre de la aplicación para la que se va a crear el esqueleto.

Nombre del paquete base: será un *textbox* en donde habrá que introducir el nombre del paquete base de la aplicación para la que se va a crear el esqueleto.

Datos de conexión: aunque para este paso no es necesario conectarse a una base de datos, rellenar ahora los datos de conexión permite al Generador crear el esqueleto ya preparado para la base de datos sobre la que vaya a trabajar la aplicación.

- Host: *textbox* donde se introduce el servidor donde se aloja la base de datos.
- Puerto: *textbox* numérico donde se introduce el puerto en el que escucha el servidor de bases de datos.
- SID: *textbox* donde se introduce el nombre (SID) de la base de datos.
- Usuario: *textbox* donde se introduce un usuario con permisos de lectura de la base de datos.
- Contraseña: *textbox* de tipo password donde se introduce la contraseña del usuario.
- Comprobar: botón que lanza una conexión con los datos introducidos hacia la base de datos, e informa de si ha podido conectar o no. Pese a que en esta etapa no es necesario conectar a la base de datos, este botón permite comprobar, en el caso de que el servidor de bases de datos esté activo, si los datos de conexión introducidos son correctos.

Parámetros: permite seleccionar mediante un *checkbox* si se utilizarán parámetros en la aplicación que se va a generar. De ser así, se activarán dos campos de texto, donde se habrán de introducir el prefijo de los nombres de los parámetros (que acabará en «_», de no ser así el Generador lo pondrá al final del nombre introducido) y la ruta donde se encuentran esos parámetros.

Tablas maestras: permite seleccionar mediante un *checkbox* si se utilizarán tablas maestras en la aplicación que se va a generar. De ser así, se activarán dos campos de texto, donde se habrán de introducir el prefijo de los nombres de las tablas maestras (que acabará en «_», de no ser así el Generador lo pondrá al final del nombre introducido) y la ruta donde se encuentran las tablas maestras.

Usuarios: permite seleccionar mediante un *checkbox* si la aplicación que se va a generar tendrá soporte para diferentes usuarios. De ser así, se activarán los siguientes campos:

- Prefijo de los nombres de usuario: *textbox* en el que se introducirá el prefijo que llevarán los nombres de usuario, seguido del carácter «_». De no ser así, el Generador lo pondrá al final del nombre introducido.
- Ruta de los usuarios: *textbox* donde se introducirá la ruta donde se encuentra los archivos de usuario.
- Grupos: *checkbox* que indica que la aplicación a generar ha de tener soporte para grupos de usuarios.
- Perfiles: *checkbox* que indica que la aplicación a generar ha de tener soporte para diferentes perfiles de usuario. Al activar esta opción, se activan los siguientes campos:
  - *Textbox* para añadir un nuevo perfil.
  - Lista que muestra los perfiles añadidos.
  - Botón de «añadir perfil», que añade el nombre del perfil que figura en el *textbox* a la lista.
  - Botón de «borrar perfil», que elimina de la lista el perfil seleccionado.

Librerías: lista de las librerías disponibles, de donde se seleccionarán las que la nueva aplicación necesite para funcionar.

Generar: botón que confirma las opciones introducidas anteriormente y genera el paquete que contendrá el esqueleto ya creado.

Retroceder: botón que cancela el proceso y devuelve al usuario a la página principal.

A.9.2. Generación de script de tablas maestras

Este apartado consta de tres etapas:

A.9.2.1. Conexión

Esta pantalla consta de los siguientes campos:

Datos de conexión:

- *Tipo de base de datos: lista para seleccionar el tipo de base de datos a la que se va a conectar (añadido posteriormente).*

- Host: *textbox* donde se introduce el servidor donde se aloja la base de datos.
- Puerto: *textbox* numérico donde se introduce el puerto en el que escucha el servidor de bases de datos.
- SID: *textbox* donde se introduce el nombre (SID) de la base de datos.
- Usuario: *textbox* donde se introduce un usuario con permisos de lectura de la base de datos.
- Contraseña: *textbox* de tipo password donde se introduce la contraseña del usuario.

Conectar: Botón que lanza la conexión contra el servidor de bases de datos según los datos introducidos.

Retroceder: botón que cancela el proceso y devuelve al usuario a la página principal.

Al pulsar el botón «Conectar», y si se puede establecer la conexión, se pasa a la siguiente etapa. En caso contrario, informa del error de conexión y devuelve al usuario a la pantalla de conexión para modificar los datos.

A.9.2.2. Selección de tablas

Una vez establecida la conexión, se presenta al usuario una pantalla con los siguientes campos:

Lista de tablas: lista doble que muestra todas las tablas contenidas en la base de datos, permitiendo seleccionar al usuario las que convengan para la generación del script de tablas maestras.

Siguiente: botón que confirma las opciones introducidas anteriormente y envía al usuario a la pantalla de selección de campos.

Retroceder: botón que cancela el proceso y devuelve al usuario a la pantalla de conexión.

A.9.2.3. Selección de campos

Tras seleccionar las tablas que interesan, el sistema nos muestra una pantalla con una lista de todas las tablas que hemos seleccionado en el paso anterior y todos los campos que las componen, así como una serie de opciones que se explican a continuación:

ID de tabla: muestra la tabla cuyos atributos aparecen a continuación en la lista.

Clase de tabla: muestra la clase (nombre del *bean*) de cada tabla.

TableName: *textbox* que permite establecer un nombre de tabla.

Default: lista de *radios* que permiten seleccionar el campo por defecto en cada tabla.

ID: muestra el ID de los campos de cada tabla.

Clase: muestra el tipo de dato que se almacena en cada campo.

Clave: indica si los datos almacenados en esa campo son claves primarias de ella o ajenas referenciando a otras.

Obligatorio: si se selecciona, impide que ese campo pueda tomar valores nulos. Si esta restricción ya está impuesta en la base de datos, el *checkbox* aparecerá tildado e inactivo.

Literal: permite establecer el literal de cada campo. Su valor por defecto será el identificador de campo en *formato Java*[4].

Listado:

- *Checkbox* de activación. El usuario lo activará si quiere que el atributo señalado sea *listable*. Al activarlo, aparecen seleccionables las siguientes opciones:
- Índice: permite establecer el orden en el que aparecerán en la lista los diferentes campos de una tabla.
- Ordenable: muestra si el listado conseguido es ordenable o no.

Buscador:

- *Checkbox* de activación. El usuario lo activará si quere que ese campo aparezca en la función buscador que aparecerá en la página del futuro proyecto.
- Índice: indica[5] el orden en el que aparecerán los resultados del buscador con los campos introducidos.
- MaxLength: permite establecer la máxima medida del campo.

Detalle:

- *Checkbox* de activación. El usuario lo activará si quiere que para el atributo seleccionado haya una vista «detalle».
- Índice: Índice: permite establecer el orden en el que aparecerán en los resultados los campos introducidos.
- MaxLength: permite establecer la máxima medida del campo.

Foreign Keys:

- *Checkbox* de activación. El usuario lo activará si quiere que la *foreign key* detectada sea tenida o no en cuenta.

[4]Ejemplo: el campo `ID_PRODUCTO_FABRICADO` tendrá como literal por defecto `idProductoFabricado`.

[5]El índice de buscador se actualiza automáticamente al tildar el checkbox de activación.

- Campo: *combo* que permitirá seleccionar qué campo de la tabla referenciada por la foreign key es el que contiene el nombre del registro que nos interesa.
- Nombre de tabla referenciada: se muestra a modo informativo el nombre de tabla referenciada por la FK presente.
- Nombre de campo referenciado: se muestra a modo informativo el nombre del campo referenciado por la FK presente.

A.9.3. Generación de código

Este apartado consta de cinco etapas:

A.9.3.1. Conexión

Esta pantalla consta de los siguientes campos:

Datos de conexión:

- *Tipo de base de datos: lista para seleccionar el tipo de base de datos a la que se va a conectar (añadido posteriormente).*
- Host: *textbox* donde se introduce el servidor donde se aloja la base de datos.
- Puerto: *textbox* numérico donde se introduce el puerto en el que escucha el servidor de bases de datos.
- SID: *textbox* donde se introduce el nombre (SID) de la base de datos.
- Usuario: *textbox* donde se introduce un usuario con permisos de lectura de la base de datos.
- Contraseña: *textbox* de tipo password donde se introduce la contraseña del usuario.

Conectar: botón que lanza la conexión contra el servidor de bases de datos según los datos introducidos.

Retroceder: botón que cancela el proceso y devuelve al usuario a la página principal.

A.9.3.2. Selección de la tabla principal

En este apartado se presentan al usuario los siguientes campos:

Lista de tablas: listado de todas las tablas que forman parte de la base de datos a la que se ha conectado, donde el usuario seleccionará la que le interese.

Siguiente: botón que lleva al usuario al siguiente paso.

Retroceder: botón que devuelve al usuario a la pantalla de conexión a la base de datos.

A.9.3.3. Selección de tablas relacionadas

En este apartado se presentan al usuario los siguientes campos:

Nombre de la tabla principal: muestra el nombre de la tabla principal, la que se ha seleccionado en el paso anterior y a la que hacen referencia las tablas de la lista que hay a continuación.

Lista de tablas relacionadas: se presenta una lista con todas las tablas que hay en la base de datos que hacen referencia a la tabla principal, es decir, que contienen claves ajenas que corresponden con la clave primaria de la tabla principal. Cada tabla irá acompañada de un *checkbox* que permitirá seleccionarla o no.

Siguiente: botón que lleva al usuario a la pantalla de selección de campos.

Retroceder: botón que devuelve al usuario al paso anterior.

A.9.3.4. Selección de campos

ID de tabla: muestra la tabla cuyos atributos aparecen a continuación en la lista.

ID: muestra el ID de los campos de cada tabla.

Clase: muestra el tipo de dato que se almacena en cada campo.

Clave: indica si los datos almacenados en esa campo son claves primarias de ella o ajenas referenciando a otras.

Obligatorio: indica si en la base de datos ese campo puede tomar valores nulos o no. Si el campo es obligatorio (*mandatory*) en la base de datos, el *checkbox* aparecerá seleccionado e inactivo. Si no lo es, podremos seleccionarlo para que en nuestra aplicación sí lo sea.

MaxLength: indica la longitud máxima que puede tomar un campo de tipo cadena de caracteres. Ofrece la posibilidad de aumentar en el programa generado la restricción que imponga la base de datos, o de establecerla si no existe.

Descripción: permite introducir el nombre (*literal*) de cada campo.

Siguiente: botón que lleva al usuario a la pantalla de selección de campos.

Retroceder: botón que devuelve al usuario al paso anterior.

Una vez seleccionados los campos a tener en cuenta y confirmadas las opciones de cada uno, el usuario introducirá las características del proyecto a crear mediante el siguiente formulario:

Nombre del paquete: *textbox* donde se introducirá el nombre del paquete que se va a generar.

Lista nombres de los *bean*s: para cada tabla (primaria y relacionadas) se ha de introducir el nombre del *bean* asociado a ellas. Constará de:

- Nombre de la clase.
- Nombre del *bean*: *textbox*.

Generar código para borrar varios *bean*s en listado: *checkbox* que permite al usuario elegir si se desea generar código para borrar varios *bean*s en listado o no.

Generar JSP de edición: *checkbox* que permite al usuario elegir si se desea generar JSPs de edición.

Generar JSP de detalle: *checkbox* que permite al usuario elegir si se desea generar JSPs de detalle.

Nombre aplicación: *textbox* donde se introducirá el nombre de la aplicación para la que se está creando el paquete.

EJBs/Clases Java: *radio* que permite al usuario elegir si se desea generar código con EJBs o con clases Java, según el servidor de aplicaciones que contenga la aplicación soporte o no EJBs. Si se seleccionan EJBs, aparecerá la siguiente opción:

EJB local/remoto: *radio* que permite al usuario elegir si se desea generar EJBs locales o remotos.

Generar: botón que confirma las opciones introducidas anteriormente y genera el archivo comprimido que contendrá el código del paquete.

Retroceder: botón que cancela el proceso y devuelve al usuario a la pantalla de resumen/opciones de campos

A continuación se presenta un esquema de los campos que contendrán los diferentes apartados de la interfaz web (figuras A.14 en la página siguiente y A.15 en la página 143):

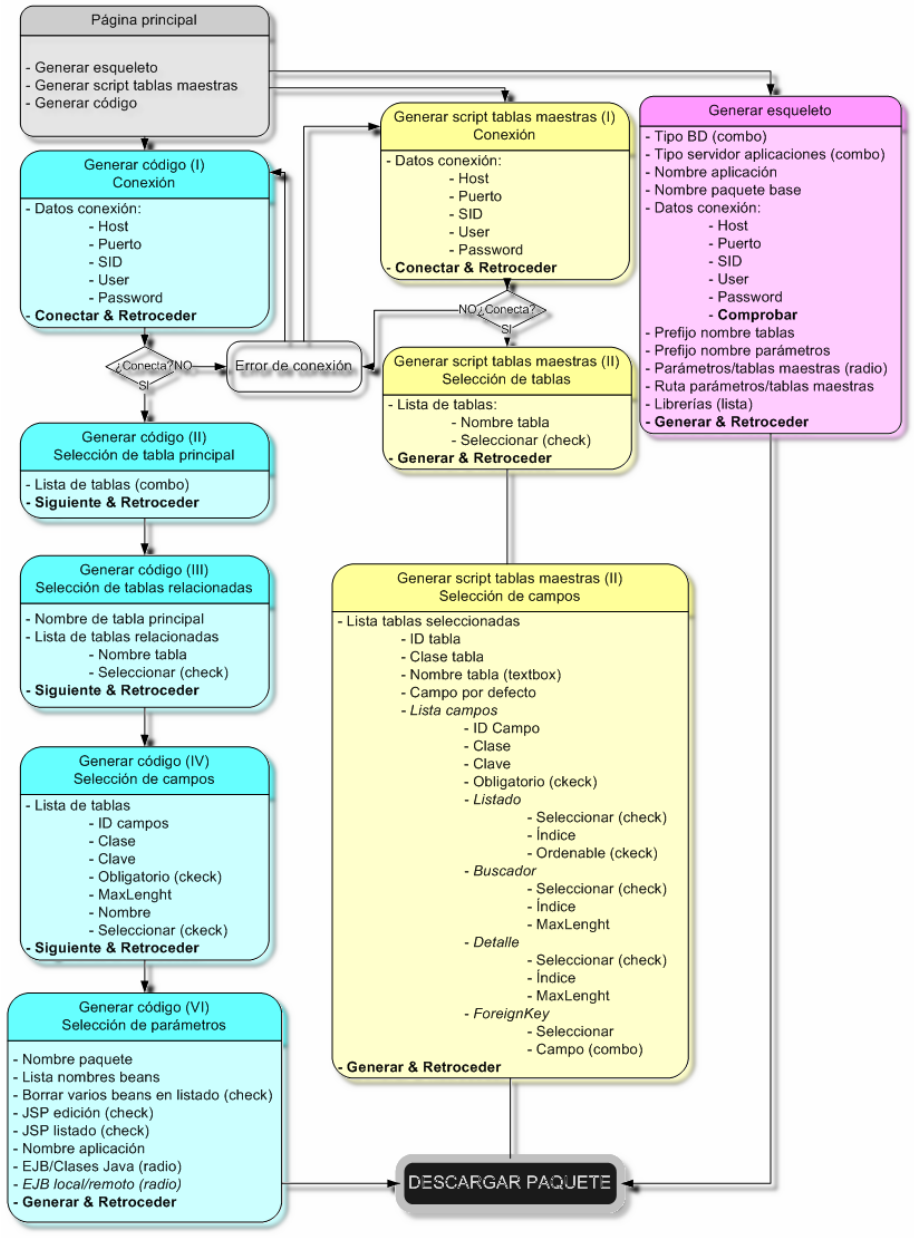

Figura A.14: Esquema de la interfaz del generador

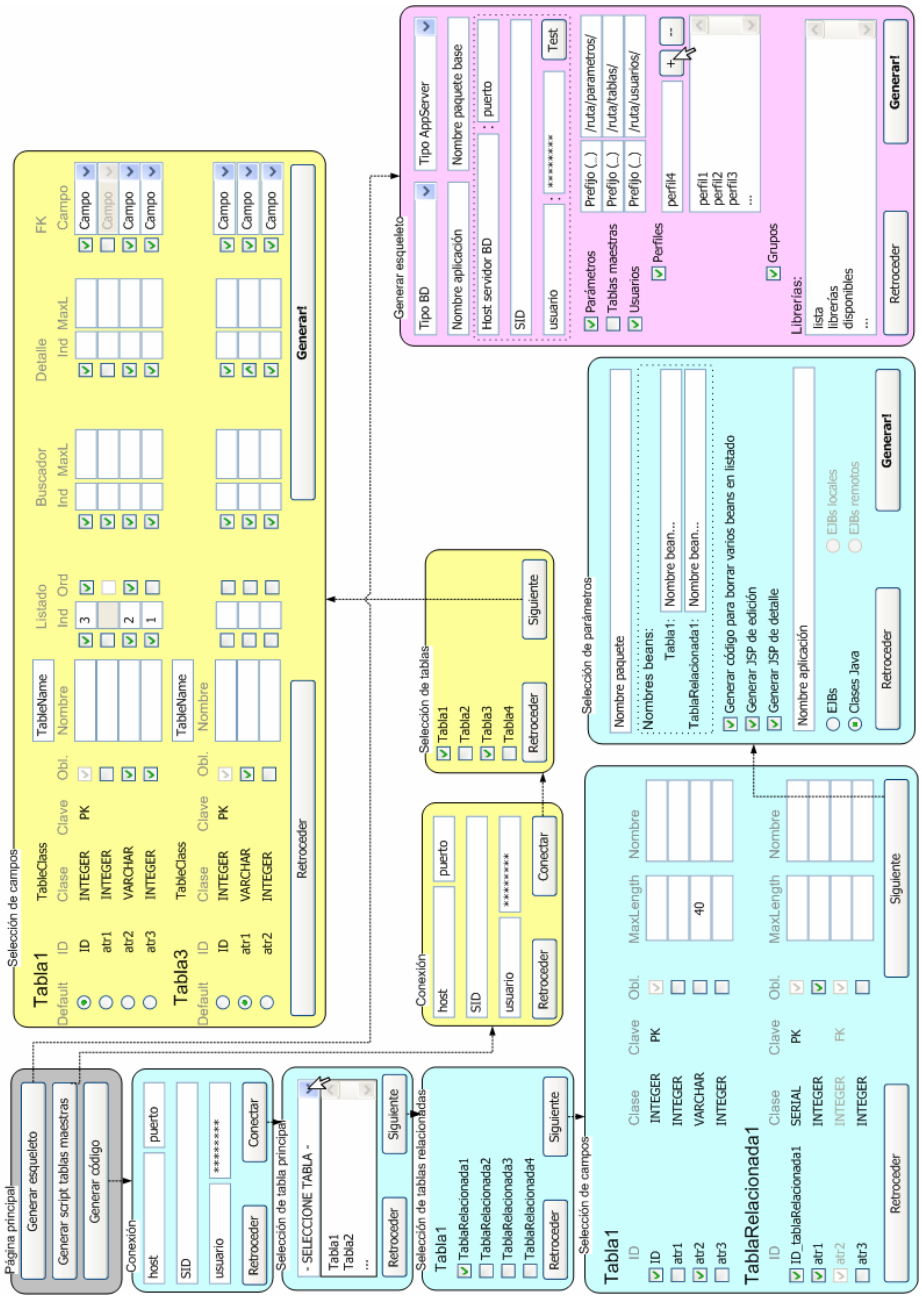

Figura A.15: Recopilación de los formularios de la interfaz del generador

A.10. Migración inicial de datos

Dado que el Generador no se apoya en ninguna base de datos propia, sino que trabaja a partir de las bases de datos de las aplicaciones que va a crear, no existe migración inicial.

Apéndice B

Combineitor, herramienta complementaria

Como ya se ha visto, en la fase de *generación de esqueleto* se proporcionan ciertos ficheros «incompletos», cuyo contenido se amplia mediante *trozos* incluídos en el paquete que se devuelve al usuario en la fase de *generación de código* de cada entidad de la base de datos.

En un principio, el mecanismo de inclusión de dichos trozos en sus archivos correspondientes se llevaba a cabo a mano, pegando el trozo en el archivo generado en el esqueleto. Este proceso es breve, no lleva más de cinco minutos, y está esquematizado en la figura B.1.

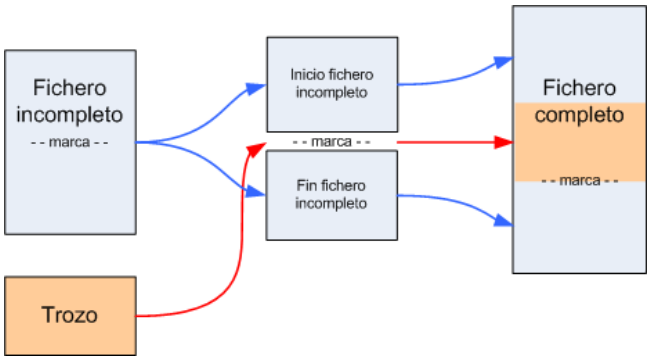

Figura B.1: Proceso de combinar un fichero a trozos.

Sin embargo, en la fase de implementación de las plantillas, se llevaban a cabo decenas de pruebas cada día para comprobar el correcto funcionamiento de las aplicaciones generadas, por lo que el tiempo perdido en completar los archivos se multiplicaba, disminuyendo enormemente la producción y ralentizando el desarrollo del proyecto. Dichos ficheros son los siguientes:

- acciones.properties

- etiquetas.properties
- InicioFilter.java
- FacadeEJB.java
- FacadeEJBBean.java
- ejb-jar.xml
- orion-ejb-jar.xml
- Scripts SQL o PL/SQL

Como solución temporal, se desarrolló un pequeño script de bash que combinaba dichos trozos automáticamente, evitando al programador la tarea de completarlos a mano. Pero tras un análisis, se decidió incorporar definitivamente este script al sistema, de forma que todos los usuarios pudieran disponer de él.

Para ello se mejoró sensiblemente el sistema de combinación, estableciendo en las plantillas de los ficheros incompletos marcas en forma de comentarios. El script analizará y reconocerá dichas marcas, insertando en su lugar los trozos correspondientes.

También se le dotó de una interfaz más amigable que la línea de comandos, optando por una basada en diálogos mediante el paquete dialog, disponible en la práctica totalidad de distribuciones GNU/Linux.

B.1. Funcionamiento

El funcionamiento de combineitor es el siguiente:

1. Tras generar el esqueleto, el usuario descomprime el contenido del paquete devuelto por Genereitor en una carpeta dentro de su directorio de trabajo. Este paquete contiene ya una copia de combineitor.
2. La etapa de generación de código devuelve un paquete, correspondiente a todos los componentes generados para la entidad de la base de datos sobre la que se trabajará.
3. El usuario copia este paquete, sin descomprimir, en el directorio donde reside el esqueleto de la aplicación.
4. Se accede por consola al directorio de trabajo donde reside el esqueleto y se otorga permiso de ejecución a combineitor[1]:

```
1 $ chmod +x combineitor
```

[1]La operación de otorgar permisos de ejecución sólo es necesario realizarla la primera vez que se ejecuta el script.

5. Tras esto, se ejecuta el script:

```
$ ./combineitor
```

Una vez lanzado el script, aparece un mensaje de confirmación, que indica a qué entidad de la base de datos corresponde el paquete que se pretende combinar, tal como se ve en la figura B.2.

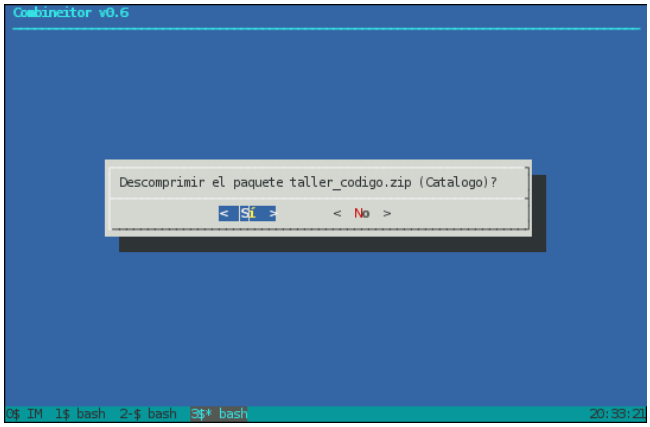

Figura B.2: Mensaje de confirmación del paquete de código de combineitor.

Tras aceptar este mensaje, se muestra una lista de los *trozos* disponibles en el paquete de código para combinar con sus correspondientes en el esqueleto, tal como muestar la figura B.3.

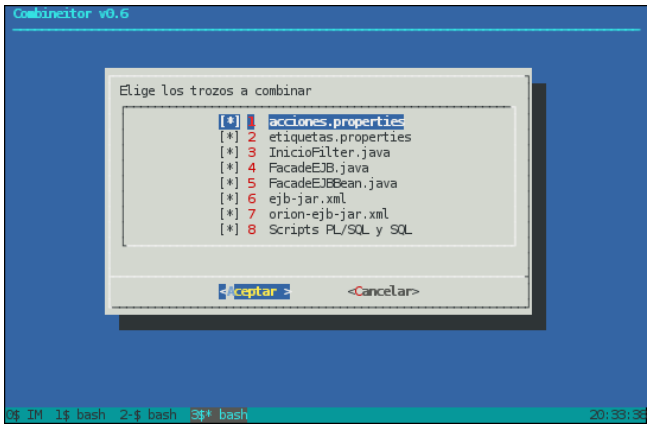

Figura B.3: Lista de selección de los ficheros a combinar.

Tras seleccionar el usuario los ficheros convenientes (que por norma general serán todos ellos, razón por la que aparecen por defecto todos seleccionados), se procede a copiar todos los archivos del paquete de código en su ubicación correspondiente dentro del esqueleto de la aplicación.

Los ficheros compuestos por trozos serán completados con el contenido de las partes del paquete de código que correspondan.

Por último, si se ha seleccionado combinar los scripts SQL o PL/SQL, se pregunta al usuario si ha habido envíos de la aplicación (figura B.4 en la página siguiente). El motivo de esta pregunta es que en la aplicación original, para montar la base de datos en los servidores del cliente se proporcionan diversos scripts SQL, organizados según su contenido, y que hay que ejecutar en orden:

- 01_TABLAS.SQL
- 02_CLAVES_AJENAS.SQL
- 03_INDICES.SQL
- 04_SECUENCIAS.SQL
- 05_VISTAS.SQL
- 06_TRIGGERS.SQL
- 07_CODIGO_00.SQL
- 07_CODIGO_01.SQL
- 08_JOBS.SQL
- 09_SINONIMOS.SQL
- 10_ROLES.SQL
- 11_GRANTS.SQL
- 12_DATOS.SQL

Si el cliente todavía no ha recibido una versión de la aplicación que se está desarrollando, los trozos de los scripts se distribuirán en estos scripts iniciales. Sin embargo, si al cliente ya se le ha entregado una versión de la aplicación, el incluír las modificaciones en estos ficheros implicaría que el cliente tendría que eliminar su base de datos y volver a ejecutar todos los scripts por orden, con la consecuente pérdida de los datos que pudiera haber añadido (o la necesidad de hacer un respaldo de la información).

Es por ésto que las modificaciones en la base de datos se proporcionan, a partir del primer envío a cliente de la aplicación, en ficheros *parche*, cuya ejecución modificará el comportamiento de la base de datos conforme los objetivos requeridos. Así, si en el diálogo de combineitor que pregunta por los envíos de la aplicación, se generará con el contenido de los trozos un fichero `XX_FECHA_DESCRIPCION.SQL`, por ejemplo `XX_20080810_ADICION_TABLA_CATALOGO.SQL`.

Tras haber realizado este último paso, la aplicación está lista para desplegar en el servidor de aplicaciones[2].

[2]Antes de desplegar la aplicación hay que ejecutar los scripts creados contra la base de datos, puesto que de otra manera no será posible que la lógica de acceso de la aplicación consulte o modifique los datos correspondientes a la nueva entidad.

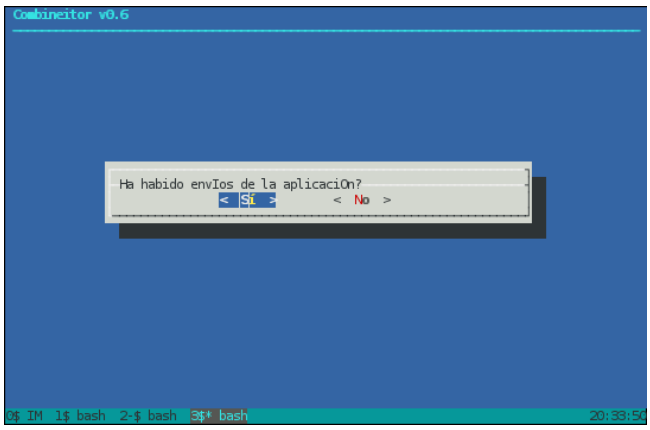

Figura B.4: Diálogo de selección de aplicación enviada o no a cliente.

B.2. Ejemplo de uso

Un ejemplo de uso, para mostrar el funcionamiento de la herramienta: el fichero ejb-jar.xml generado con el esqueleto tiene esta apariencia:

```
1  <!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
       2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
2  <ejb-jar>
3     <enterprise-beans>
4        <session>
5           <description>Session Bean ( Stateless )</description>
6           <display-name>TallerFacadeEJB</display-name>
7           <ejb-name>TallerFacadeEJB</ejb-name>
8           <home>com.comex.taller.bl.ejb.TallerFacadeEJBHome</home>
9           <remote>com.comex.taller.bl.ejb.TallerFacadeEJB</remote>
10          <ejb-class>com.comex.taller.bl.ejb.TallerFacadeEJBBean</ejb-class>
11          <session-type>Stateless</session-type>
12          <transaction-type>Bean</transaction-type>
13 <!-- ################################### -->
14 <!--genereitor:insertarTrozo1-->
15 <!-- ################################### -->
16       </session>
17 <!-- ################################### -->
18 <!--genereitor:insertarTrozo2-->
19 <!-- ################################### -->
20    </enterprise-beans>
21    <assembly-descriptor>
22 <!-- ################################### -->
23 <!--genereitor:insertarTrozo3-->
24 <!-- ################################### -->
25    </assembly-descriptor>
26 </ejb-jar>
```

Se observan en las líneas 13, 17 y 22 las trazas que utilizará combineitor para saber dónde insertar los trozos. Se etiquetan con el literal «genereitor» para indicar a los programadores que son trazas generadas automáticamente, que no son comentarios de un desarrollador, y que por lo tanto no se pueden quitar. De lo contrario, combineitor no realizaría su tarea de forma correcta.

El trozo de dicho fichero para una entidad *catálogo* contendrá lo siguiente:

```
1  <ejb-local-ref>
2     <ejb-ref-name>ejb/local/CatalogoEJB</ejb-ref-name>
3     <ejb-ref-type>Session</ejb-ref-type>
4     <local-home>com.comex.taller.bl.ejb.CatalogoEJBLocalHome</local-home>
5     <local>com.comex.taller.bl.ejb.CatalogoEJBLocal</local>
6  </ejb-local-ref>
7
8  <!--genereitor:finTrozo1-->
9
10 <session>
11    <description>Session Bean ( Stateful )</description>
12    <display-name>CatalogoEJB</display-name>
13    <ejb-name>CatalogoEJB</ejb-name>
14    <local-home>com.comex.taller.bl.ejb.CatalogoEJBLocalHome</local-home>
15    <local>com.comex.taller.bl.ejb.CatalogoEJBLocal</local>
16    <ejb-class>com.comex.taller.bl.ejb.CatalogoEJBBean</ejb-class>
17    <session-type>Stateful</session-type>
18    <transaction-type>Bean</transaction-type>
19 </session>
20
21 <!--genereitor:finTrozo2-->
22
23 <container-transaction>
24    <method>
25        <ejb-name>CatalogoEJB</ejb-name>
26        <method-name>*</method-name>
27    </method>
28    <trans-attribute>Required</trans-attribute>
29 </container-transaction>
```

En este fichero, las marcas de corte están en las líneas 8 y 21. El fin del último trozo no es necesario indicarlo, puesto que corresponde con el fin de fichero.

Una vez realizada la combinación, el resultado será el siguiente:

```
1  <!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
       2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
2  <ejb-jar>
3     <enterprise-beans>
4        <session>
5           <description>Session Bean ( Stateless )</description>
6           <display-name>TallerFacadeEJB</display-name>
7           <ejb-name>TallerFacadeEJB</ejb-name>
8           <home>com.comex.taller.bl.ejb.TallerFacadeEJBHome</home>
9           <remote>com.comex.taller.bl.ejb.TallerFacadeEJB</remote>
10          <ejb-class>com.comex.taller.bl.ejb.TallerFacadeEJBBean</ejb-class>
11          <session-type>Stateless</session-type>
12          <transaction-type>Bean</transaction-type>
13 <!-- ################################## -->
14          <ejb-local-ref>
15             <ejb-ref-name>ejb/local/CatalogoEJB</ejb-ref-name>
16             <ejb-ref-type>Session</ejb-ref-type>
17             <local-home>com.comex.taller.bl.ejb.CatalogoEJBLocalHome</local-
                   home>
18             <local>com.comex.taller.bl.ejb.CatalogoEJBLocal</local>
19          </ejb-local-ref>
20 <!--genereitor:insertarTrozo1-->
21 <!-- ################################## -->
22       </session>
23 <!-- ################################## -->
24       <session>
25          <description>Session Bean ( Stateful )</description>
26          <display-name>CatalogoEJB</display-name>
27          <ejb-name>CatalogoEJB</ejb-name>
28          <local-home>com.comex.taller.bl.ejb.CatalogoEJBLocalHome</local-home
                >
29          <local>com.comex.taller.bl.ejb.CatalogoEJBLocal</local>
30          <ejb-class>com.comex.taller.bl.ejb.CatalogoEJBBean</ejb-class>
31          <session-type>Stateful</session-type>
32          <transaction-type>Bean</transaction-type>
```

```
33          </session>
34    <!--genereitor:insertarTrozo2-->
35    <!-- ################################## -->
36       </enterprise-beans>
37       <assembly-descriptor>
38    <!-- ################################## -->
39          <container-transaction>
40             <method>
41                <ejb-name>CatalogoEJB</ejb-name>
42                <method-name>*</method-name>
43             </method>
44             <trans-attribute>Required</trans-attribute>
45          </container-transaction>
46    <!--genereitor:insertarTrozo3-->
47    <!-- ################################## -->
48       </assembly-descriptor>
49    </ejb-jar>
```

Vemos que los trozos se han insertado correctamente en su sitio, quedando separadas por comentarios las partes generadas por el esqueleto y las generadas por el código. También se mantienen las marcas de inserción, de forma que el proceso de combinación se pueda repetir al añadir otra entidad de la base de datos a la aplicación.

B.3. Motivos de la elección de **bash** para su implementación

Como ya se ha comentado, esta herramienta auxiliar se concibió en un principio como mera ayuda al desarrollo de las plantillas de Genereitor, por ello se eligió un lenguaje sencillo, rápido y del que ya se tuviera experiencia previa.

Cuando se tomó la decisión de incluírlo en Genereitor como herramienta complementaria, se planteó portar el código a otro lenguaje más eficiente y que soportase multiplataforma, pero se desestimó la idea, puesto que la totalidad de los integrantes del grupo de desarrollo de aplicaciones J2EE trabajan bajo sistema operativo GNU/Linux, por lo que no era necesario contar con independencia de plataforma.

Asímismo, no es un útil imprescindible, ya se ha comentado que el proceso que realiza combineitor puede ser ejecutado por el programador en menos de cinco minutos.

Apéndice C

Tecnologías utilizadas

C.1. JavaScript

JavaScript es un lenguaje de scripts (guiones) generalmente utilizado para el desarrollo de páginas web, ofreciendo funcionalidades en el lado del cliente (client-side)[1], sin necesidad de establecer conexión con el servidor.

Es un lenguaje de programación interpretado[2] y su sintaxis es similar a la de Java o C. Es un lenguaje orientado en objetos basado en prototipos, y dispone de herencia, disponible mediante clonación de objetos existentes, que hacen la función de prototipos.

Debido a que JavaScript se ejecuta directamente en el navegador del usuario, en lugar de en un servidor remoto, responde instantáneamente a las acciones del usuario, mejorando la experiencia del usuario.

Las características principales de JavaScript son:

- Es un lenguaje basado en objetos (arrays asociativos[3]. ampliados con prototipos).
- Soporta la sintaxis de programación estructurada de los lenguajes más comunes (C, Java). Hace distinción entre *expresiones* y *declaraciones*.
- Permite evaluar expresiones en tiempo de ejecución. Es decir, acepta el paso como parámetro de una cadena de caracteres, que interpretará como una expresión y, tras evaluarla, devolverá un resultado.

[1] Aunque inicialmente JavaScript se utilizaba únicamente para operaciones en el lado del cliente, aparecieron plataformas que soportan su uso en el lado del servidor (LiveWire JavaScript), mediante motores tales como Rhino, JScript o SpiderMonkey. Una guía de referencia elaborada por Sun para el trabajo con JavaScript del lado del servidor se puede consultar en http://docs.sun.com/source/816-6408-10/contents.htm.

[2] No requiere compilación.

[3] Un array asociativo es un tipo de dato abstracto, compuesto por una colección de claves únicas y una colección de valores, donde cada clave está asociada con un valor. Las operaciones básicas de un array asociativo son add (asignar una nueva clave a un nuevo valor), reasign (modificar el valor referenciado por una clave), remove (desvincular un valor a una clave) y lookup (encontrar el valor asociado a una clave).

- Es un lenguaje *débilmente tipificado*, puesto que no es necesario declarar el tipo de datos en la asignación de variables, y la conversión de un tipo a otro se realiza de forma automática. A esta característica se le conoce como *duck typing*. El siguiente fragmento de código es perfectamente válido:

```
1  var variable1 = 2;
2  variable1 =+ 3;        //2 + 3, variable1 es de tipo numérico
3  alert(variable1);      //devolverá 5
4  variable1 = "Cadena de caracteres";
5  alert(variable1);      //devolverá "Cadena de caracteres"
6  variable1 = true;      //variable1 es de tipo booleano
7  if(variable1) {
8          alert("Verdadero");
9  } else {
10         alert("Falso");
11 }
```

- Las funciones son objetos, y como tales tienen propiedades y pueden interactuar con otros objetos.
- Usa prototipos en lugar de clases para definir las propiedades de los objetos.
- No distingue entre *funciones* y *métodos*. Una función puede se llamada como método.
- Soporta expresiones regulares.

El principal inconveniente de JavaScript es que no es accesible. Es por ésto que no es recomendable su uso en páginas que requieran un nivel aceptable de accesibilidad, o en caso de que se decida usarlo, es necesario ofrecer una alternativa bajo el tag `<noscript>` en la página. Por ejemplo, si se ofrece un menú desplegable que funciona con JavaScript, se ofrecerá también el menú desplegado u otra alternativa para los navegadores que no soporten JavaScript.

C.1.1. Alternativas

Como ya se ha dicho, si la aplicación requiere de un nivel aceptable de accesibilidad es necesario proveer de alternativas al uso de JavaScript, puesto que en caso de utilizar navegadores antiguos o lectores de pantalla para personas con problemas de visión, es probable que no sea posible ejecutar dichos guiones.

Muchas de las funciones que se implementan en el lado *cliente* con JavaScript se pueden modelar en el servidor, mediante JSP, PHP, CGI y tecnologías similares. El problema de delegar al servidor tareas que se pueden hacer en cliente es evidente: se requieren más conexiones entre ambos lados para realizar la misma tarea, lo que repercutirá en el rendimiento final de la aplicación.

C.2. JSP

JavaSever Pages es una tecnología de Java que permite generar dinámicamente documentos HTML, XML o de otros tipos, en respuesta a una petición de

un cliente web. Permite el uso de código Java y proporciona algunas acciones predefinidas que pueden ser incrustadas en el contenido estático.

La sintaxis JSP añade *tags*, al estilo de XML, llamadas *acciones JSP* y que son utilizadas para invocar a las funcionalidades provistas por el uso de JSP. Además, se ofrece la posibilidad de crear nuevos *tags*, extendiendo la librería de los ya existentes (provistos por HTML o XML). Es un método de ampliar las funciones de un servidor web manteniendo la independencia de plataforma. Los JSP son compilados para convertirse en Java Servlets en el momento en el que un cliente hace la primera petición. Las sucesivas peticiones se verán satisfechas con más rapidez, ya que no será necesario volver a compilar el JSP.

C.2.1. Alternativas

La alternativa más inmediata al uso de JSP es ASP, de Microsoft. No obstante, JSP ofrece las siguientes ventajas con respecto a su rival:

C.2.1.1. Plataforma e independencia del servidor

JSP sigue la filosofía de Java de *write once, run anywhere*. Su característica multiplataforma nos proporciona independencia sobre el sistema operativo que debe llevar el servidor que aloje el contenedor donde se almacenarán, compilarán y ejecutarán los JSP. Por el contrario, ASP está limitada a arquitecturas basadas en tecnología de Microsoft.

Así, JSP se puede ejecutar en los sistemas operativos y servidores más comunes, como son Apache, Netscape o Microsoft IIS, mientras que ASP sólo tiene soporte nativo para servidores Microsoft IIS y Personal WebServer, ambos de Microsoft.

C.2.1.2. Proceso de desarrollo abierto

El API de JSP se beneficia de la extendida comunidad Java existente, por el contrario el desarrollo de ASP depende únicamente de Microsoft.

C.2.1.3. Tags

Mientras que tanto JSP como ASP usan una combinación de tags y scripts para crear paginas web dinámicas, la tecnología JSP permite a los desarrolladores crear nuevos tags. Así los desarrolladores pueden crear sus tags de forma personalizada, y no depender de los scripts. Esto proporciona una alta reutilización de código.

C.2.1.4. Java

La tecnología JSP usa Java para crear contenido de forma dinámica, mientras que ASP usa VBScript o Jscript. Java es un lenguaje mas rápido, potente y

escalable que los lenguajes de script. Asímismo, JSP puede utilizar JavaScript para tareas de lado del cliente.

C.2.1.5. Mantenimiento

Las aplicaciones que usan JSP tiene un mantenimiento más fácil que las que usan ASP.

- Los lenguajes de script están bien para pequeñas aplicaciones, pero no encajan bien para aplicaciones grandes. Java es un lenguaje estructurado y es más fácil de construir y mantenimientos grandes como aplicaciones modulares.
- La tecnología JSP hace mayor énfasis en los componentes que en los scripts, esto hace que sea más fácil revisar el contenido sin que afecte a la lógica o revisar la lógica sin cambiar el contenido.
- La arquitectura EJB encapsula la lógica de acceso a BD, seguridad, integridad transaccional y aislamiento de la aplicación.
- Debido a que la tecnología JSP es abierta y multiplataforma, los servidores web, plataformas y otros componentes pueden ser fácilmente actualizados o cambiados sin que afecte a las aplicaciones basadas en la tecnología JSP.

C.2.2. Inconvenientes

Uno de los inconvenientes con respecto a otras tecnologías similares es que tiene una curva de aprendizaje más pronunciada. Aprender a programar en JSP requiere más esfuerzo, sobre todo si no se tiene experiencia anterior en lenguajes de programación, que ASP.

C.3. Ant

Apache Ant es una herramienta utilizada para la realización de tareas mecánicas y repetitivas, especialmente durante el proceso de compilación y construcción de aplicaciones. Es similar a make, pero está implementado usando Java, requiere la máquina virtual de Java y se integra perfectamente con proyectos en dicho lenguaje.

La principal diferencia con make es que utiliza XML para describir el proceso de construcción y sus dependencias, mientras que make utiliza su propio formato Makefile. Los scripts XML utilizados por ant llevan por nombre `build.xml`.

Ant es software de código abierto, y está liberado bajo la *Apache Software License.*

C.3.1. Ventajas

La principal ventaja de este sistema con respecto a sus predecesores es la portabilidad.

Mientras que make recibe las acciones necesarias para realizar una acción como órdenes del intérprete de comandos propio del sistema operativo sobre el que se ejecuta, ant resuelve esta situación proveyendo por él mismo gran cantidad de funciones, lo que garantiza que permanecerán idénticas en todos los sistemas operativos.

Por ejemplo, en un Makefile para Unix, la orden *borrar recursivamente un directorio* sería:

```
rm -rf directorio/
```

La orden `rm` es propia del intérprete de Unix, si intentamos ejecutar make con esta orden en un entorno Windows, no funcionará. Sin embargo, un script *build.xml* para ant que hiciera la misma función que el anterior sería:

```
<?xml version="1.0"?>
<project name="borrar" default="borrarDirectorio" basedir=".">
    <target name="borrarDirectorio"/>
              <delete dir="directorio"/>
         </target>
</project>
```

Y este script funcionaría en cualquier sistema que contase con una máquina virtual Java instalada, puesto que no depende de las órdenes del intérprete de comandos.

C.3.2. Desventajas

Pese a las ventajas anteriormente nombradas, ant adolece de una serie de inconvenientes:

- Al ser una herramienta basada en XML, los scripts ant deben ser escritos en XML. Esto no sólo es una barrera para los nuevos usuarios, que tienen que aprender este lenguaje, sino también un problema para los proyectos muy grandes, cuando los scripts comienzan a hacerse muy grandes y complejos. Esto es un problema común a todos los lenguajes que utilizan XML, pero la granularidad de las tareas de ant significa que los problemas de escalabilidad no tardan en aparecer.
- La mayoría de las antiguas herramientas (`<javac>`, `<exec>`, `<java>`) tienen malas configuraciones por defecto y valores para las opciones que no son coherentes con las tareas más recientes. Pero cambiar estos valores supone destruir la compatibilidad hacia atrás. Esto también sucede al expandir propiedades en una cadena o un elemento de texto, las propiedades no definidas no son planteadas como error, sino que se dejan como una referencia sin expandir.

- No es un lenguaje para un flujo de trabajo general, y no debe ser usado como tal. Tiene reglas de manejo de errores limitadas y no dispone de persistencia de estado, por lo que no puede ser usado con confianza para manejar una construcción de varios días.

C.3.3. Alternativas: Maven

Una alternativa clara al uso de ant que superara muchas de las limitaciones de las que éste adolece es el uso de maven.

Maven aplica patrones a la estructura del proyecto, lo que debería promover la productividad, facilitar la comprensión de la estructura del proyecto y reducir el coste de trabajar con un nuevo proyecto, ya que la estructura será idéntica.

Normalmente cuando se comienza un nuevo proyecto en el que se va a usar ant, se copian los scripts de un proyecto ya existente y se modifican para adaptarlos al nuevo[4]. Maven soluciona esto mediante el uso de patrones que indican cómo ha de ser un proyecto Java y convenciones sobre la configuración.

Maven, como herramienta de gestión de proyectos, proporciona funciones de:

- Compilación.
- Documentación (e.g. Javadoc).
- *Reporting* sobre datos del proyecto.
- Gestión de dependencias.
- SCM[5] y *releases*.
- Distribución de la aplicación a servidores locales o remotos.

Este sistema distingue entre diferentes ciclos de desarrollo. Hay 3 ciclos de desarrollo predefinidos: *default*, *clean* y *site*. El primero (*default*) gestiona el desarrollo del proyecto, el segundo (*clean*) elimina los artefactos producidos por el primero y el tercero (*site*) se encarga de la documentación y el sitio web del proyecto.

C.3.4. Motivos de la elección de ant

Pese a que es evidente que existen alternativas mejores a ant, se ha decidido mantener esta herramienta por diversas razones. He aquí las más importantes:

[4]En realidad, este proceso de copiar-pegar-adaptar no se tiene por qué llevar a cabo, ya que una de las características de Genereitor es precisamente ofrecer estos scripts adaptados al proyecto que se está generando, de forma automática y totalmente funcional desde el principio de la implementación del proyecto.

[5]Control de versiones de forma transparente, sin importar si debajo se usa CVS, GIT, SVN o cualquier otro.

Requisitos del cliente. Como ya se ha comentado a lo largo de la memoria, Comex depende de los requisitos de Aragonesa de Servicios Telemáticos para la elaboración de los proyectos que este organismo le encarga, y al ser uno de los mayores clientes, se adoptan dichos requisitos para todas las aplicaciones J2EE desarrolladas. Así, el uso de ant viene impuesto por AST, por lo que es más cómodo realizar las tareas de gestión de todos los proyectos con esta herramienta que introducir varias que, realizando la misma función, su funcionamiento sea diferente.

Generación automática de scripts. Uno de los inconvenientes que se ha planteado respecto a ant es lo engorroso que es generar los scripts de compilación de forma correcta para aplicaciones de cierta envergadura. En este caso en particular este inconveniente no es tal, ya que Genereitor proporciona dichos scripts de forma automática.

Integración con IDEs. Aunque es posible utilizar Maven con cualquier IDE, los más usados (NetBeans, Eclipse y Oracle JDeveloper) proporcionan por defecto soporte para ant[6], por lo que la integración es instantánea.

C.4. XSLT

Extensible Stylesheet Language Transformations, transformaciones de lenguaje extensible de hojas de estilo. Es un lenguaje basado en XML, cuyo cometido es la transformación de documentos XML en otros documentos (tanto XML como otros formatos), a partir de una serie de datos. XSLT es un estándar de W3C[7], y junto a XPath forman parte de XSL.

Además de la transformación de documentos XML, XMLT ofrece también otras funciones:

- Añadir y eliminar elementos a un árbol XML.
- Añadir y eliminar atributos a un árbol XML.
- Reorganizar y mostrar elementos.
- Mostrar u ocultar determinados elementos.
- Encontrar o seleccionar elementos específicos.

En el proceso de transformación intervienen dos archivos, el de datos, que contendrá todos los nodos de información formateada en XML, y la plantilla XSLT, que será la base del documento resultante de la combinación de ambos. El proceso de combinación se ilustra en la figura C.1 en la página siguiente.

Un ejemplo sencillo de fichero de datos puede ser el siguiente:

[6]De hecho, al generar un nuevo proyecto en NetBeans o importar uno ya existente, el propio entorno de programación proporciona automáticamente un build.xml para compilar y desplegar dicho proyecto.

[7]La especificación de XSLT se puede consultar en http://www.w3.org/XML/.

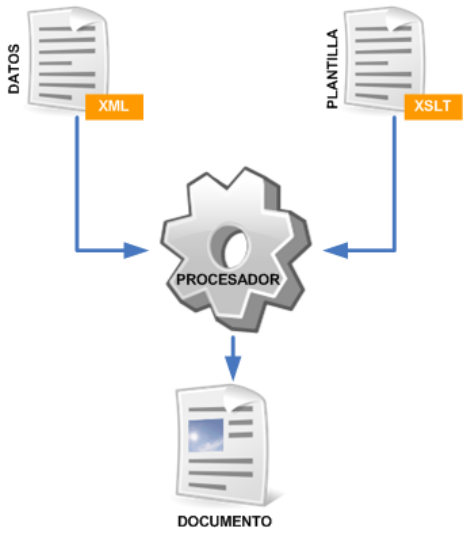

Figura C.1: Proceso de combinación de XSLT

```
<?xml version="1.0"?>
<personas>
   <persona>
      <nombre> Dante </nombre>
      <telefono> 666555444 </telefono>
      <edad> 18 </edad>
   </persona>
   <persona>
      <nombre> Randal </nombre>
      <telefono> 666777888 </telefono>
      <edad> 22 </edad>
   </persona>
</personas>
```

Vemos que los datos se hallan agrupados en *nodos*, conformando una estructura de árbol.

Y una plantilla muy básica puede ser:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0
    ">
<xsl:output method="text" omit-xml-declaration="yes" indent="yes"/>
   <xsl:template match="/">
Hay <xsl:value-of select="count(personas/persona)"/> personas.
   <xsl:for-each select="personas/persona">
Nombre: <xsl:value-of select="nombre"/>.
Edad: <xsl:value-of select="edad"/>.
   </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Al combinar ambos ficheros, obtendremos el siguiente resultado:

```
Hay 2 personas.
Nombre: Dante.
Edad: 18.
Nombre: Randal.
Edad: 22.
```

Como se puede comprobar, el formato de salida en este caso es texto simple, sin ningún formato. Esto destaca la capacidad de XSLT de crear casi cualquier tipo de archivos a partir de los datos recopilados en el fichero de datos.

En Genereitor se generan con XSL diversos tipos de ficheros: clases Java, archivos XML, páginas JSP, incluso ficheros de configuración propios de Oracle JDeveloper. Pero sus posibilidades no se terminan ahí. Es común el uso de esta tecnología para elaborar a partir de plantillas y de datos introducidos en un formulario web documentos PDF.

C.4.1. Ventajas

Con todo lo visto, se pueden destacar las siguientes ventajas de esta tecnología:

- No asume un único formato de salida de documentos.
- Permite manipular de muy diversas maneras un documento XML: reordenar elementos, filtrar, añadir, borrar, etc.
- Permite acceder a todo el documento XML.
- XSLT es un lenguaje XML.

También es importante el hecho de que existen diferentes procesadores XSLT, como son Saxon, Xalan, LotusXSL o Unicorn, por lo que podemos adecuar el proceso respecto a nuestros requisitos tanto de rendimiento como de licencias.

C.4.2. Inconvenientes

Asímismo, también se han de mencionar diversos inconvenientes que plantea el uso de XSLT:

- Su utilización es más complicada que la de un lenguaje de programación *convencional*, y su sintaxis es algo incómoda.
- Consume una cantidad sustancial de recursos, que dependerán de la forma en que esté elaborada la plantilla XSLT, pero que para transformaciones de cierto volúmen de datos o plantillas con flujos *complicados*, puede ser elevada.

- La potencia del lenguaje es limitada, su abanico de opciones para controlar el flujo se limita a bucles condicionales (`if`, `for-each`, etc), por lo que la realización de plantillas complejas puede ser poco intuitiva. Además, al no permitir la declaración de funciones, no es posible la reutilización del código dentro de una misma plantilla.

No obstante y pese a las carencias listadas, hay que tener en cuenta que XSLT es un lenguaje de tratamiento de datos, y no de propósito general, por lo que no ha de utilizarse como tal.

C.4.3. Alternativas

Hay varias alternativas que permitirían la generación de documentos a partir de plantillas.

Una de ellas es DSSSL[8], el predecesor de XSLT, que ofrece características similares para SGML[9] (en lugar de XML). No obstante, este sistema, aunque es muy parecido a XSLT, está siendo reemplazado por este último.

Otras opciones serían JavaScript Style Sheets (JSSS), Formatted Output Specification Instance (FOSI) o Sintactically Awesome StyleSheets (SASS), pero no son tecnologías estándar.

Por último, también se puede utilizar para realizar transformaciones de plantillas la tecnología JSP, que proporcionaría la potencia del lenguaje Java y permitiría simplificar la elaboración de las plantillas más complicadas. No obstante, se ha preferido utilizar XSLT por la eficiencia en las plantillas sencillas, que serán las más comunes. Genereitor cuenta con más de 150 plantillas, que si se tuvieran que implementar en archivos JSP supondrían una carga importante, tanto en tiempo de compilación como en ejecución.

C.5. Javadoc

Javadoc es una utilidad de Sun Microsystems para la generación de documentación de APIs en formato HTML a partir de código fuente Java.

Los comentarios Javadoc están destinados a describir, principalmente, clases y métodos. Como están pensados para que otro programador los lea y utilice la clase (o método) correspondiente, se decidió fijar al menos parcialmente un formato común, de forma que los comentarios escritos por un programador resultaran legibles por otro. Para ello los comentarios Javadoc deben incluir unos indicadores especiales, que comienzan siempre por '@' y se suelen colocar al comienzo de línea.

Lo que hace esta herramienta es extraer los comentarios Javadoc contenidos en el programa Java indicado y construir con ellos ficheros .html que puede servir como documentación de la clase.

[8] *Document Style Semantics and Specification Language*, lenguaje de especificación y semántica de estilo de documentos, especificado por el estándar ISO/IEC 10179:1996.

[9] *Standard Generalized Markup Language*, lenguaje estándar de marcas generalizado (ISO 8879:1986).

Javadoc es el estándar de la industria para documentar clases de Java. La mayoría de los IDEs los generan automáticamente.

C.5.1. Alternativas

La principal alternativa a Javadoc es Doxygen, que utiliza un método similar para generar la documentación, pero es compatible con diversos lenguajes, entre ellos C++, C, Java, Objective-C, Python, IDL, PHP, C# y D. Asimismo las interfaces generadas por Doxygen son más agradables a la vista.

Doxygen, al igual que Javadoc, es multiplataforma.

El motivo de la elección de Javadoc es que es el estándar de Sun para generar documentación. Además, al estar mucho más extendido, los IDEs más comunes ofrecen soporte nativo para este formato de documentación, e incluso son capaces de crearla automáticamente.

Bibliografía

[1] The Java EE 5 Tutorial:
http://java.sun.com/javaee/5/docs/tutorial/doc/index.html

[2] Sun Enterprise BluePrints: Guidelines, Patterns, and code for end-to-end Java applications:
http://java.sun.com/blueprints/enterprise/index.html

[3] Nivel Doble-A de Conformidad con las Directrices de Accesibilidad para el Contenido Web 1.0
http://www.w3.org/WAI/WCAG1AA-Conformance

[4] Sun Developer Network: J2EE & JavaServlet Pages Technology:
http://java.sun.com/products/jsp/

[5] Sun Developer Network: Development with JSP and XML:
http://java.sun.com/developer/technicalArticles/xml/WebAppDev3/index.html

[6] JavaServlet Scriptlets, Richard G. Baldwin:
http://www.developer.com/tech/article.php/626401

[7] Ley de la Comunidad Autónoma de Aragón 7/2001, de 31 de mayo, de creación de la Entidad Pública Aragonesa de Servicios Telemáticos:
http://www.lexureditorial.com/boe/0106/11883.htm

[8] Java EE y Web 2.0:
http://www.programania.net/diseno-web/JavaScript/java-ee-y-web-20/

[9] Arquitectura J2EE:
http://www.proactiva-calidad.com/java/arquitectura/index.html

[10] Estr@tegia Magazine, Año 3, Edición nº 52, Sección Tecnología:
http://www.e-estrategia.com.ar

[11] *Interfaces gráficas en Java*, Micael Gallego Carrillo y otros, ed. Ramón Aceres, 2005.

[12] Adobe, Centro de desarrolladores: Introducción a XSL:
http://www.adobe.com/es/devnet/dreamweaver/articles/xsl_overview.html

[13] JSP. Custom Tags, etiquetas a medida:
http://www.programacion.com/java/articulo/customtags/

[14] Filtros y Servlets en Java:
www.samelan.com/oscargonzalez/doc/java_filters.pdf